UNICA

UNIVERSITÀ
DEGLI STUDI
DI CAGLIARI

Ph.D. DEGREE IN
Electronic and Computer Engineering

Cycle XXXV

Optimizing Neural Networks for Embedded Edge-Processing Platforms.

ING-INF/01

Ph.D. Student: Paola Busia

Supervisor Prof. Paolo Meloni

Final exam. Academic Year 2021/2022
Thesis defence: February 2023 Session

To my Family,

Contents

1 Introduction **3**

2 State of the Art **7**

2.1 Edge-processing Platforms 8

2.2 CNN model optimization 9

2.2.1 Network compression 9

2.2.2 Hardware-Aware Neural Architecture Search 10

2.2.3 Hardware cost modeling 12

2.2.4 Neural Architecture Search for KWS 15

2.3 Emerging Neural Networks 17

3 Optimizing the design space exploration: the ALOHA design flow **19**

3.1 ALOHA CNN design flow 23

3.1.1 Dataset Management Utility 23

3.1.2 Selection Procedure 24

3.1.3 Evaluation tools 25

3.1.4 Deployment 27

3.2 CNN selection procedure implementation 27

3.2.1 Fast implementation 27

3.2.2 Accurate Implementation 30

3.2.3 Selection time 31

3.3 Experimental Results 33

3.3.1 Search Space definition 33

3.3.2 Preliminary Pre-processing exploration 34
3.3.3 Use-case 1 35
3.3.4 Use-case 2. 38
3.3.5 Selection's quality assessment. 42
3.3.6 Closing remarks. 42

4 Hardware-aware performance estimation: the ALOHA method 45

4.1 Background 47
4.1.1 The computational tensor of a CNN layer. 47
4.1.2 OPS-based Performance prediction 49
4.1.3 Roofline Model 50
4.2 ALOHA estimation method 51
4.3 ALOHA platform Model 52
4.4 ALOHA evaluation Procedure 58
4.4.1 Computational tensor generation 59
4.4.2 Computational tensor refinement 60
4.4.3 Computational tensor analysis 62
4.5 CNN metric aggregation 64
4.6 Experimental Results 67
4.6.1 Layer-level accuracy 67
4.6.2 Impact on energy consumption estimation 70
4.6.3 Impact on NAS 71
4.6.4 Impact of aggregation on prediction accuracy 77
4.6.5 Closing remarks 81

5 Emerging Neural Networks: Optimal Transformer Design 83

5.1 Problem definition 84
5.2 Detector 86
5.2.1 EEGformer. 86
5.2.2 CNN on raw EEG signal. 89
5.2.3 CNN on pre-processed input features. 89

5.3 Assessment on CHB-MIT dataset 90
5.4 Deployment 94
5.4.1 Deployment on Apollo4 94
5.4.2 Deployment on GAP: exploit parallelism 94
5.4.3 Closing remarks 95

6 Conclusions **99**

Abstract

The design of a Convolutional Neural Network suitable for efficient execution on embedded edge-processing platforms requires reconciling accuracy and efficiency requirements. Several research efforts have translated this task into the iterative search of Pareto-optimal points satisfying multiple objectives, but a step forward is still needed to assist the developer in this complex task. In this thesis, we summarize the key challenges of edge-oriented design into three main topics.

As a first point, the size of the design space is so big it makes any full exploration unfeasible, thus, effective practices to limit the exploration time without compromising its outcome are needed.

Additionally, edge-processing platforms are highly heterogeneous and often endowed with specialized accelerators, therefore the prediction of the hardware performance of the candidate design points requires a certain degree of platform awareness.

Finally, the recent advancements in the neural network domain have uncovered emerging models and intelligence mechanisms, whose success has encouraged their optimization for deployment at the edge. The transformer represents a remarkable example.

In this thesis, we present our contribution to these relevant design challenges. First, we describe an efficient design flow to jointly evaluate several design parameters, referring to a Keyword Spotting task targeting a commercial micro-controller for its evaluation. We provide a fast exploration strategy, requiring around 30 hours and resulting in state-of-the-art accuracy within the defined storage constraints. We further consider a more accurate exploration strategy, allowing us to refine the performance evaluation during the search process with an additional characterization time.

As a second contribution, we present an accurate, flexible, and easy-to-use estimation method for the most relevant hardware performance metrics, such as latency, energy consumption, and throughput, to be integrated into an automated design flow and enable modeling the network execution on the most typical families of edge-processing devices. The proposed method improves the prediction accuracy of state-of-the-art approaches of comparable complexity, not requiring access to direct on-hardware measurements during the exploration process, and improves by up to 4× the predictability of hardware-aware Neural Architecture Search.

As the last contribution, we present a tiny transformer model for long-term epilepsy monitoring, suitable for real-time seizure detection on low-power health-monitoring devices. The assessment of its performance shows accuracy metrics well-aligned with the state of the art, obtainable with as low as 13.7ms inference time and 0.19mJ energy consumption per inference.

Chapter 1

Introduction

Convolutional Neural Networks (CNNs) have represented for years the state of the art in the image classification [1, 2, 3, 4] and object detection [5] tasks, thus becoming one of the standard approaches to computer vision in various fields, from autonomous driving [6] to medical imaging [7, 8]. Their success is not strictly limited to image processing, as they have been also successfully applied, among other fields, to speech recognition [9, 10] and medical signals processing [11, 12].

The ability to learn complex data representation is obtained through complex architectures stacking multiple computational intensive operators, dominated by convolutions, which require the storage of a significant amount of trainable parameters. To give an example, the ResNet-50 architecture employs 26 MB of parameters and requires 4.1 billion of Floating-Point operations (FLOPs) [13].

Despite such high computational and storage requirements, the growing success of CNNs has encouraged their deployment on a variety of different processing systems, including the edge-processing domain. This deployment scenario introduces tight resource constraints, requiring careful optimization of the classifier design and representing a very challenging task. As the range of CNN-based applications grows, there is a need for efficient design solutions to assist the system developer in defining the optimal CNN architecture to meet the accuracy and performance constraints.

Despite the ongoing research efforts, edge-oriented design still presents two key challenges. The first one is represented by the huge design space of architectural and processing

possibilities: the CNN structure is obtained as the composition of multiple stages, each applying different transformations to the input data; the different combinations of parameters (shape and operand) for each stage, as well as their number and sequence, result in a wide range of different models, reaching different levels of accuracy and representing a different computational workload. Moreover, the system design should also consider any data preparation stage, as well as any possible optimization step which could be applied to a candidate CNN model, to reduce its computational or storage requirement.

At the same time, the constrained deployment scenario demands a tight trade-off between the complexity and the accuracy of the classifier. However, the edge-processing domain is a composed set of hardware targets, exploiting computing units based on different architectural solutions, thus the comparison of the efficiency of two alternative CNN models is a non-trivial task, where the specific characteristics of the target hardware have to be taken into account. These main points can be summarized as the need to provide an efficient design exploration strategy, and the need to enable a certain degree of platform-awareness.

Furthermore, the continuous advancements in the neural network subject have introduced new operands and architectural solutions, which enrich the CNN baseline of new features and capabilities. The transformer represents a relevant example, as it has recently challenged the supremacy of CNNs in some of the most relevant application fields. The interest in this emerging model has grown to include edge deployment, thus the optimization of its architecture for the target domain has become a relevant subject.

In this thesis, we mean to contribute on these challenges. Our contributions can be outlined in three points:

- the definition of an effective design exploration strategy, comprehensively considering the effects of tuning the available design parameters on the performance and efficiency of edge implementation, within a limited exploration time [14];
- the development of an accurate estimation method to predict the on-hardware efficiency of a candidate design point on a given target platform[15];
- the application of hardware-aware design to a transformer architecture [16]; the optimization process resulting in the proposed architecture does not exploit the design

flow presented in [14], but it shows that very similar concepts apply to this family of networks.

Here we briefly describe the organization of the following pages. In Chapter 2 we revise the state of the art of edge-processing platforms and neural networks optimization strategies, focusing on automated design based on Neural Architecture Search (NAS) and considering platform awareness. In Chapter 3 we present our design procedure, describing two different implementations representing different trade-offs between exploration effort and the quality of the resulting design. We refer to the ALOHA framework (Software framework for runtime-Adaptive and secure deep Learning On Heterogeneous Architectures[1]), which provided the context and reference for the contribution presented in this chapter. In Chapter 4 we describe our ALOHA estimation method for the prediction of platform-aware metrics, aiming to enable hardware awareness in NAS-based design targeting the embedded domain. In Chapter 5 we present an optimized transformer model designed for epilepsy monitoring on low-power devices. Finally, the conclusions are summarized in Chapter 6.

[1]https://www.aloha-h2020.eu/

Chapter 2

State of the Art

In recent years there has been a growing interest in enabling artificial intelligence on tiny smart devices, providing near-sensor processing and thus reducing the power consumption of battery-powered devices, easing the bandwidth requirements of a cloud-based processing system, and ensuring the privacy and security of the user's sensitive data [17]. All these aspects are of particular relevance in some application fields, including industrial applications [18] and health-monitoring [19].

In the following, we start with a brief review of the state of the art of efficient edge-processing platforms specialized for the neural networks' workload, in Section 2.1. Then, we focus in Section 2.2 on the network model optimization, considering network compression (Section 2.2.1) and automated design based on Neural Architecture Search (Section 2.2.2). In Section 2.2.3 we revise the literature of the existing approaches enabling platform awareness in the NAS process, whereas we summarize in Section 2.2.4 the most relevant works exploiting NAS for the design of an efficient Keyword Spotting (KWS) classifier, as we will reference a KWS use case for the definition and evaluation of our design strategy. Finally, we discuss in Section 2.3 some relevant works dealing with the transformer model and its current application scenarios.

2.1 Edge-processing Platforms

The recent trend to enable artificial intelligence capabilities on low-power tiny devices, fitting the size and energy requirements of the edge-computing paradigm, has encouraged the design of an abundance of specialized computing platforms for the efficient execution of neural networks (NNs). The work of [20] provides a very recent survey on the most popular hardware platforms for edge NN-inference. This rich landscape includes several solutions, representing an effort toward adjusting the computational power of the target devices to the typical workload of neural networks, dominated by convolutions and matrix multiplications [21]. In the following, we list some of the solutions available, such as:

- application-specific accelerators, like Google Edge TPU [22] and NVIDIA Deep Learning Accelerator (NVIDLA) [23], optimized to reach high performance in the execution of tensor processing operations;
- systems-on-chip, such as Intel Movidius [24], a highly optimized video processing unit based on 16 cores and a dedicated hardware engine for neural networks acceleration, and Qualcomm's Snapdragon [25], targeting gaming applications and embedding a specialized accelerator;
- tiny GPUs, such as NVIDIA Jetson [26];
- several development boards based on micro-controller units (MCUs), such as ST Sensortile [27] and Ambiq Apollo4 [28], as well as on parallel computing clusters, like Greenwaves GAP processors [29], supporting Single Instruction Multiple Data;

Along with commercial devices, several academic research prototypes have been proposed, such as the Eyeriss flexible application-specific accelerator [30], or FPGA-based computing engines, like the one presented in [31] and NEURAGHE [32]. The computing platforms can support different data precision: for example, NVIDLA natively supports various data types, including 2-, 4-, 8-, 16-bit integer representations, and 16-, 32- and 64-bit floating point representation [23].

In most cases, high performance is obtained thanks to parallel computing on dedicated accelerators. In order to exploit at best the performance enabled by these specialized comput-

ing units, efficient dataflows optimizing the use of the hierarchical memory systems and the communication resources need to be implemented [30]. Due to this reason, the wide range of hardware solutions is accompanied by a family of specialized software libraries enabling efficient access to the computing resources: representative examples are cuDNN [33], targeting GPUs, CMSIS-NN [34], specifically optimized for the ARM Cortex-M family of cores (like the one embedded in the SensorTile), and the PULP-NN library [35], targeting PULP-based clusters (like the GAP processors).

2.2 CNN model optimization

Alongside the design of more efficient computing architectures, a parallel line of research deals with the optimization of deep network structures into more portable models, which remains a primary concern to fit the low-power requirements of several application scenarios, as well as the size of local storage resources close to the computing engines. The constraints on the memory footprint are especially hard when tiny MCUs are targeted, motivating the studies related to more lightweight, yet accurate and efficient, network architectures. Popular examples of handcrafted tiny network models targeting classification and object-detection on the edge-domain are SqueezeNet [36], reaching the 80.3% top-1 accuracy on the ImageNet dataset [37] for image classification with 50× fewer parameters than the comparable AlexNet [1], the MobileNet family [38], reaching up to 75% accuracy with 5.4M parameters, and TinyYolo [39], requiring 8.9 MB and reaching 34% mAP on the COCO dataset [40].

2.2.1 Network compression

Along with the time-consuming design-from-scratch approach, compression techniques reducing the size of common network models have captured great interest. The rationale of network compression is that neural networks are highly redundant: quantization exploits the robustness deriving from such redundancy to reduce the precision of the parameters representation, and consequently the memory requirements of the network model. The groundbreaking work of [41] showed how the combination of connections pruning and data quantization could be exploited to reduce by $35\times$ the storage requirements of AlexNet [1] without affecting

its accuracy on the ImageNet dataset [37].

A very recent survey on the current approaches to quantization is provided in [42]. The main concept is to map the real values to a reduced precision interval, based on a scaling factor and a zero point. A relevant work for the selection of the representation interval is represented by PACT [43], addressing the challenge of quantizing the activations through an optimal scaling factor learned during training. The authors show that full-precision accuracy can be obtained with as low as 4-bit representation, considering the performance of AlexNet, ResNet-18, and ResNet-50 on the ImageNet dataset [37]. This line of research is called Quantization-Aware Training, allowing the recovery of the accuracy loss resulting from the reduced precision with an additional computational cost. Despite the relevant results obtained in [43], with commonly available methods lossless quantization can be easily achieved up to 8-bit representation, whereas recovering the accuracy drop resulting from more aggressive strategies is still not straightforward.

2.2.2 Hardware-Aware Neural Architecture Search

Even with the support of these common compression methods, the design of a network architecture targeting a given application on a specific hardware platform is a complex task, requiring much application- and NN-related expertise. The work of [44] introduced NAS as an automated design procedure, defined as an iterative search process and based on the definition of a *search space*, a *search strategy*, and a *performance evaluation* model. Exploiting a Recurrent Neural Network (RNN) as a controller, to first generate CNN models as design points in the search space and then evaluate them based on their expected accuracy, the authors improved the state-of-the-art CNN on the CIFAR-10 dataset [45], selecting the best network architecture among 12800 model configurations. Moving from this first single-objective approach [46, 47], hardware-aware NAS (HW-NAS) [48] includes the on-hardware performance metrics in the search loop, considering the Pareto-optimal points in terms of validation accuracy and an additional hardware-related cost function. In the following, we discuss some of the most relevant works on HW-NAS.

The authors of [49] studied a compound scaling method, to obtain new versions of known CNN architectures based on the hardware requirements by jointly scaling the depth, width,

and resolution of the baseline network. When applied to state-of-the-art networks, like MobileNet and ResNet, compound upper scaling allows obtaining up to 5% accuracy improvement. After applying NAS optimizing accuracy and OPS, they select a family of networks, the EfficientNets, which reduce by around an order of magnitude the number of OPS of state-of-the-art models with comparable accuracy. As OPS count does not allow for an accurate performance estimation on the target hardware [50], other approaches consider direct on-hardware measurements during the exploration process: the authors of [51] consider direct hardware measurements of latency on target mobile devices, resulting in the design of MNasNet, reaching 75.2% ImageNet accuracy with 1.8× speedup over MobileNetV2 [52]; in [53] peak power and average energy consumption measurements are included in the reward function to evaluate the candidate networks during the exploration.

The literature reports many other examples of works relying on HW-NAS for optimal network design, considering different performance metrics estimations [54, 55, 56, 57, 58, 59]. The subject of hardware performance modeling will be further discussed in Section 2.2.3.

As an application-specific NAS could require a huge amount of iterations, several solutions have been proposed for the search optimization, exploiting one-shot training and weight sharing [59, 60]. The OFA framework [59] allows defining the design space as a super-network, which is trained once to optimize all its possible sub-networks, representing the candidate design points. Considering several deployment scenarios, the authors exploit a single training procedure to select the optimal network targeting CPUs, GPUs, and FPGA accelerators, based on an accuracy predictor and on latency look-up-tables (LUTs). They obtained up to 4% ImageNet accuracy improvement over MobileNetV3. Finally, to keep up with the wide adoption of reduced-precision CNNs, some automatic design flows also consider compression through quantization, as an optimization objective along with network topology selection [61, 62].

In this thesis, considering the lesson of [59], we present an efficient design space exploration to address the selection of multiple design knobs in a target-oriented CNN optimization problem. As we selected a KWS task as a target use case, we will further discuss the comparison with NAS oriented to the KWS problem in Section 2.2.4.

Category	***Methods***	***Metric***	***Accuracy***	***Re-usability***	***Modularity***
OPS	[49]	*latency*	*low*	*very high*	✗
	[63]	*latency, energy, memory*			
Roofline	[64]	*latency*	*medium/ low*	*high*	✗
Specialized analytical methods	[65]	*latency, energy, memory*	*medium/ high*	*low*	✗
	[56, 66]	*latency,energy*			
	[54, 55, 67]	*latency, energy throughput*		*medium*	
Measurements	[57]	*latency*	*high*	*very low*	✗
	[53]	*latency, energy*			
Look-up tables (LUT)	[59]	*latency*	*high*	*low*	✗
	[68]	*latency*	*high*	*medium/ low*	
ML	[69]	*latency, energy*	*high*	*very low*	✗
	[70]	*latency*			
ALOHA	***this thesis Ch.* 4**	***latency, energy throughput***	***medium/ high***	***high***	✓

Table 2.1: Comparison of methods for evaluation of platform-aware CNN metrics

2.2.3 Hardware cost modeling

Platform awareness represents a key point in most of the recent works dealing with an optimal design based on NAS. Table 2.1 provides an overview and comparison of the methods commonly exploited in the literature for the evaluation of platform-dependent CNN metrics. The state-of-the-art estimation methods, listed in Column 2, are grouped into general hardware-performance modeling categories, indicated in Column 1. The examined works are associated with the list of platform-aware metrics they are able to capture, given in Column 3, and compared in terms of their accuracy (Column 4), degree of re-usability (Column 5), and level of modularity (Column 6).

A high re-usability is an important quality metric, limiting design-time overheads despite the ever-increasing number and diversity of CNNs and target hardware platforms, and refers to how many different ones can be successfully modeled with the considered method. To give an example, the operations count (OPS) category (Row 2) has very high degree of re-usability

because it is easily adaptable to a variety of CNNs and hardware platforms without requiring any changes. On the other hand, the methods based on machine learning (ML) predictors (Rows 12 to 13) exhibit very low re-usability because, once designed, the predictors are only applicable to a specific set of explored CNNs and a specific target platform, and if that set is changed they must be completely redesigned.

The modularity item, in Column 6, refers to the modeling power of the method, considering whether it can account for the modular structure of the platform, including the distribution of the layers over the different processing units in a heterogeneous platform, and their execution schedule. Based on the typical scheme adopted in common Deep Learning (DL) frameworks, such as Pytorch [71] or TensorFlow [72], CNN inference is usually sequential. Thus the platform accelerators receive the workload layer-by-layer [73]. However, as suggested in [17], efficient edge-processing can be achieved with different scheduling choices [74, 75, 76], improving the throughput and energy consumption, and thus requiring consideration when the hardware-aware metrics are estimated.

In the following, we will briefly comment on and compare the considered modeling categories, referring, for brevity, to latency estimation. As already mentioned, estimating inference latency based on the network OPS is a simple and highly reusable method. However, the obtained predictions are often inaccurate [50, 51], introducing a significant error margin between the predicted latency and the one measured on the target platform. When the design targets applications with severe timing and resource constraints, as is common in autonomous driving [77] or object recognition on drones [78], this would result in the need to replace the selected CNN model.

We indicate as Roofline methods (Row 4) those evaluating latency based on the well-known Roofline model [79]. These methods consider the impact of memory access in addition to the number of OPS required by the network, providing a more precise latency evaluation. However, the estimation accuracy is still not sufficient to correctly model the hardware performance of some CNN models on particular hardware platforms.

Rows 5 to 7 refer to specialized analytical methods producing latency estimations based on exact hardware platform representations. The increased estimation accuracy is obtained at the expense of a low re-usability. To give some examples, in [54, 55, 56] a precise roofline-based

model is exploited for FPGA co-design, whereas [67] relies on the MAESTRO model [80] for ASIC co-design. If different hardware platforms are targeted, such as CPUs or GPUs, these models can't be exploited or would require adequate adaptation.

In Rows 8 and 9 we list studies including direct on-hardware latency measurements in the design process. Based on a similar approach, works in Rows 10-11 access LUTs collecting measured latency values for single layers, or blocks, to evaluate the overall network latency. This solution ensures the best accuracy, however, it requires a large number of measurements [57, 53], or a narrow exploration based on a small LUT.

Finally, ML predictors (Rows 12 to 13), such as neural networks or regression models, can be very accurate, but they need to be trained on a large number of measurements and provide very low re-usability. Some examples of required profiling time are reported in Table 2.2. We assume an average inference time of 15ms and evaluate the data collection time of the considered works, which would be over 1 hour for [81], and over 18 hours in [57]. This is a very soft hypothesis, as [70] reports almost 2 weeks of data collection time. In general, training has to be repeated for each target platform, although some approaches suggest a training procedure considering a range of targets (e.g. [69] evaluates 447 different GPU configurations, while [81] suggests training a single network for predicting performance on multiple hardware).

In Chapter 4 of this thesis, we propose a platform-aware evaluation model to be efficiently included in the HW-NAS loop, able to provide realistic predictions of the critical performance metrics, like latency, energy consumption, and throughput. Being based on abstract high-level platform specifications, it can be applied to a wide range of hardware platforms, without requiring repeated measurements.

Method	*Data Collection time* $n_sample * t * N_{avg}$	*Training time*
[81]	$75000 * t * 5$	*(300 epochs)*
[70]	$80000 * t * N_{avg}$	*1h (1000 epochs)*
[57]	$90000 * t * 50$	*20min (150 epochs)*
[69]	$447 * 108 * t * N_{avg}$	*not specified*

Table 2.2: The required profiling time for the evaluation methods based on ML models is described as the sum of two major components: the time required to acquire the training data, and the time required to perform the training procedure. The Data Collection time is expressed as the product of 1) the number of samples evaluated, n_sample, 2) their execution time on the target hardware, t, and 3) the number of times each measure is repeated to obtain an accurate value, N_{avg}.

2.2.4 Neural Architecture Search for KWS

		Quantization		*Pre-processing*		
	Hardware metric	***Levels***	***Cross exploration***	***Parameters***	***Cross exploration***	***Exploration description***
[61]	*OPS footprint*	*8bit*	✗	*n. frames*	✓	✗
[62]	*OPS latency*	*up to 1bit*	✗	*not applicable*	*not applicable*	✓
[82]	*OPS*	✗	✗	✗	✗	✓
[83]	✗	✗	✗	✗	✗	✓
this thesis Ch. 3	*latency footprint energy*	*up to 4bit*	✓	*feature type n.features n. frames*	✓	✓

Table 2.3: Comparison with state-of-the-art works on NAS targeting KWS.

KWS utilities are typical examples of applications to be deployed on power-constrained edge devices, often having limited storage resources. Thus the design of new workload-efficient network architectures, either handcrafted [84] or resulting from NAS [61, 82, 83, 62], was encouraged. We will discuss in the following some relevant examples of the latter, which

provide a comparison reference for the design strategy presented in Chapter 3 to efficiently address the hardware-oriented design of a KWS system.

The state of the art is summarized in Table 2.3: the main works, listed in Column 1, are compared in Column 2 based on the metrics captured by their degree of platform awareness. Columns 3 and 4 refer to the quantization subject: we indicate the range of representation precision explored and whether the exploration of the CNN topology and the quantization policy is combined. Columns 5 and 6 report similar information about the pre-processing scheme, considering the most common speech features, namely Mel energies and Mel-Frequency Cepstral Coefficients (MFCC). Finally, Column 7 highlights the works providing methodological guidelines for the exploration. We briefly comment on the referenced studies in the following.

The work of [61] compares different network models obtained through NAS, considering CNNs, depth-wise separable CNNs (DS-CNNs), RNNs, and deep neural networks (DNNs). The exploration targets MCUs embedding ARM Cortex-M Processors of different sizes, defined as Small, Medium, and Large, integrating hardware awareness as a set of constraints on the number of OPS and the memory footprint. The pre-processing scheme assumes feature extraction based on MFCC, where the number of frames constituting the spectrogram is subject to exploration, resulting in a 49x10 (49 time-frames and 10 features) resolution for every evaluated model. Quantization is performed only on the finally selected model, up to 8-bit representation. The focus of the work is not on providing design guidelines, as it aims to present the well-known state-of-the-art DS-CNN architectures, reaching higher accuracy than the corresponding CNN candidates.

The authors of [62] exploit differentiable NAS based on OPS evaluation to design a CNN model reaching 95.6% accuracy, with 75.7KB of parameters and 13.6 MOPS. It is applied to raw audio files, leveraging learned parameterized *SincConv* functions executed as a first stage. Thus, this pre-processing approach can not be described according to the scheme proposed in Columns 5 and 6 in Table 2.3. Also in this case, quantization is only applied to the selected CNN model, thus it is not cross-explored with the topology: the best performing quantized version exploits 2.51-bit weights and 2.91-bit activations, reaching 93.76% accuracy.

OPS is again the reference hardware evaluation metric in [82]. This work describes an exploration methodology based on two stages: a first fixed-budget training phase, and a second

refinement step on the Pareto-optimal candidates, based on hyperparameters subject to exploration. The provided guidelines do not address the cross-exploration of the pre-processing scheme or the quantization policy, considering pre-processing based on MFCC and resulting in 40x32 feature resolution. The best-performing CNN reaches 95.1% accuracy.

As a last reference, we consider [83], presenting a model based on depth-wise separable and dilated convolutions which reaches 97.2%. Until recently, this was the state-of-the-art for convolution-based KWS models. It requires pre-processing based on MFCC, resulting in 101x40 input features. Such pre-processing scheme is pre-defined when the NAS exploration is performed.

To the best of our knowledge, as summarized in Table 2.3, ours is the first approach to provide methodological guidelines on how to efficiently combine the exploration of the optimal CNN topology, pre-processing scheme, and quantization level, considering hardware-aware performance evaluations. We believe that the impact of feature extraction and input resolution on the final accuracy and overall performance motivates our perspective, as both the inference time and storage resources are affected by these design choices. To refine the OPS indirect complexity metric into a more accurate execution time estimation, easily comparable with a maximum constraint, we exploit a latency prediction model of the target platform. Furthermore, the current trend [85] shows the importance of including quantization within the NAS exploration. We mean to extend such benefits to the KWS field.

2.3 Emerging Neural Networks

In the last years, the landscape of neural networks has evolved far beyond the simple CNNs, to include more complex and irregular models enforcing new intelligence mechanisms.

Nowadays, the transformer has gained momentum since its first appearance in the work of [86]. The attention mechanism, representing the core of the transformer, was first applied to machine translation tasks, providing state-of-the-art results thanks to a general understanding of the context.

Recent works have applied the same mechanism to other fields. The most renowned example is represented by the Vision Transformer [87], proposing a transformer architecture

applied to the image classification problem. Ever since, less conventional application fields have been studied, such as audio processing [88], and medical signal processing [89, 90, 91, 92]. This spread to include new use cases has contributed in pushing toward edge deployment also for transformer-based models [93, 89], requiring specific quantization strategies [94].

In Chapter 5 of this thesis, we finally consider the optimization of a transformer model targeting the epilepsy monitoring problem on the wearable domain, based on the typical storage constraints of tiny MCUs. The presented use-case allows to highlight the design knobs available for hardware-oriented optimization and the similarities with the CNN domain. Although the presented exploration was not conducted with the tools introduced in Chapter 3, the advantages of such a design approach could be exploited also for transformers, once the support for the specific operands is integrated.

Chapter 3

Optimizing the design space exploration: the ALOHA design flow

One of the main challenges of edge-oriented design is the huge amount of design parameters deeply affecting the performance of the system, whose exploration would be enormously time-consuming, thus requiring efficient exploration strategies. A well-suited example for the discussion of platform-aware design and optimization of efficient CNN-based system implementation is represented by the KWS task. It consists of a simple speech recognition problem, requiring the classification of a limited set of instructions[95], or the detection of few wake words, as in smart home devices, where it is coupled with more complex audio processing systems running on the cloud. We consider this use case as a reference for two main reasons:

1. the various applications relying on KWS provide multiple deployment scenarios, targeting a very wide range of processing platforms and hardware architectures, which should be precisely characterized to enable efficiency evaluations within the design flow;

2. KWS classification based on CNNs usually exploits feature-extracting techniques, representing additional design choices to be considered.

The general composition of a KWS system is highlighted in Figure 3.1: the recorded audio samples are streamed to a pre-processing stage, responsible for extracting representative features, which are provided to a CNN-based classification stage and thus assigned to a set of classes corresponding to the keywords to be recognized. Restricting the classifier space to

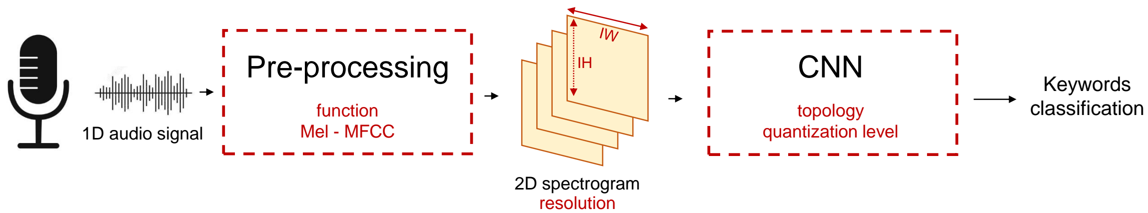


Figure 3.1: KWS system overview.

the CNN domain, an effective system design exploration should involve several parameters, impacting its performance on the target hardware:

- the feature-extracting function - we assume as reference the most common pre-processing functionality in literature, which converts the audio stream into a bi-dimensional representation of the power spectrum of the acquired audio over time, namely Mel energies and Mel-Frequency Cepstral Coefficients (MFCC);
- the resolution of the extracted features;
- the CNN topology;
- the CNN quantization level.

The vast design space resulting from all their possible combinations cannot, in general, be fully explored. In this thesis, we present a design flow to efficiently address the design of platform-specific applications, considering the KWS reference use case and targeting deployment on a tiny Micro-Controller Unit (MCU). Our proposed strategy allows to reach near-optimal solutions based on the evaluation of a restricted number of network candidates. The backbone of optimization tools is provided by the ALOHA framework, which is the result of a research project with several contributing partners, aiming to facilitate the design and deployment of efficient, yet accurate, CNN models on a desired target embedded platform [96]. Based on a combined effort with the research team of Santer Reply SpA, we defined an efficient design flow combining the tools in ALOHA, able to:

- *consider target-awareness*: predicting the effects of the different design choices on target-dependent metrics such as latency, footprint, or energy consumption, based on a model-based evaluation tool whose estimations are included in the optimization process;

- enable the combined *cross-exploration of the data pre-processing, the CNN topology, and the quantization*, as the combination of these design choices impacts the classification accuracy and the hardware efficiency.

Rather than in the introduction of one specific tool, the contribution analyzed in this chapter lies in the definition of an effective strategy, based on an HW-NAS approach, to exploit the feedback provided by the set of available evaluation tools and define a restricted set of network architectures to which the most time-consuming refinement steps should be applied. Nonetheless, some of the tools were specifically extended to provide the results disclosed in this chapter, as is detailed in the description of Figure 3.2.

Based on the design effort, we define two different versions of the design flow. The first one implements a *fast selection* procedure, detecting at-least-sub-optimal solutions with a limited exploration time. The second one involves a more precise characterization phase considering the accuracy drop connected to quantization: we refer to it as *accurate selection*.

In the following, after a brief description of the reference platform, we start with a general overview of the ALOHA framework, provided in Section 3.1, introducing the tools available for network optimization and performance evaluation. The proposed design strategy, configured as the fast and accurate implementations of our hardware-aware CNN selection procedure, is described in Section 3.2. Finally, experimental results are presented in Section 3.3. As most of the state-of-the-art works, the classification task refers to the Google Speech Commands dataset [95], and involves 10 of the 35 classes provided: ”Yes”, ”No”, ”Up”, ”Down”, ”Left”, ”Right”, ”On”, ”Off”, ”Stop” and ”Go”, plus the additional classes ”silence” and ”unknown”.

SensorTile

As an example target of our design-flow application, we consider a tiny microcontroller platform, developed by STMicroelectronics: the SensorTile. It is an IoT module, equipped with a digital microphone, and embedding an 80 MHz ARM Cortex-M4 32-bit low-power micro-

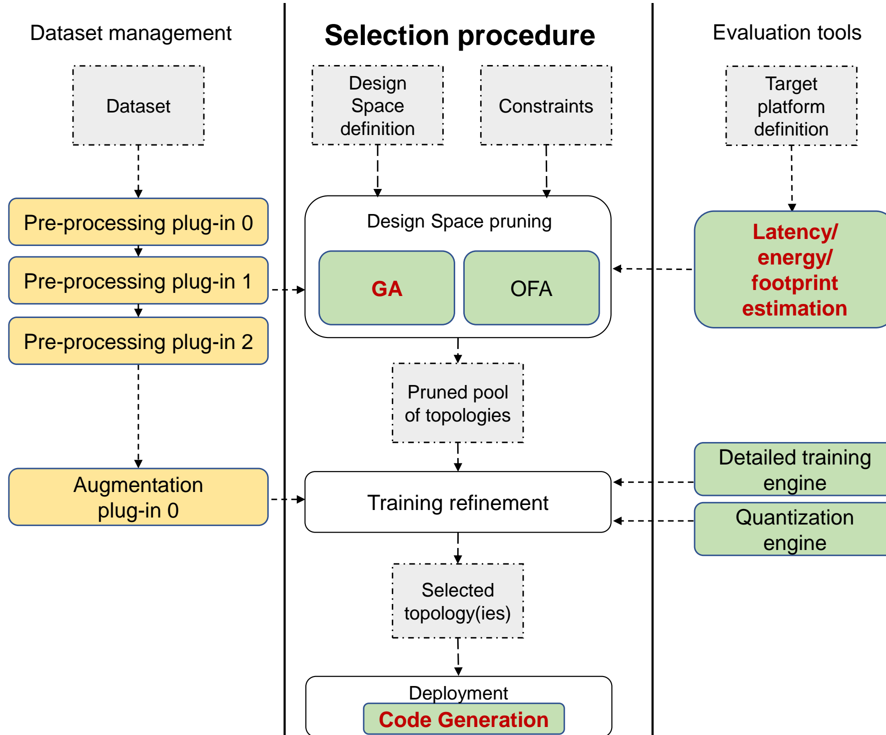


Figure 3.2: Selection procedure for target-oriented CNN design exploiting the ALOHA framework. The main stages of the evaluation procedure are depicted as white boxes, whereas the required inputs to the various stages of the exploration are represented as grey boxes. Yellow boxes represent the dataset management utilities, whereas green boxes represent the tools available on the framework for the evaluation of the candidate network models. We highlighted in red the tools specifically extended to enable the exploration described in the following sections.

controller, accessing a 96kB SRAM, and 1MB FLASH memories, posing strict storage constraints. The system architecture exploits a Real-Time lightweight Operating System (RTOS), providing support for multi-threading, and scheduling of the different application tasks on defined timings. For efficient CNN execution, we relied on the CMSIS-NN library [34], specifically developed to target this family of processors.

3.1 ALOHA CNN design flow

The ALOHA framework exploits NAS to address CNN design as an iterative selection process. Figure 3.2 depicts an overview of the framework organization. The optimization process starts from the following list of inputs:

- a reference dataset - a collection of data to be manipulated with the desired set of pre-processing operations;
- a platform description - defining the hardware resources, exploited to obtain reliable hardware-related performance metrics;
- a set of constraints - defined based on the platform's description and the application requirements and translated into a maximum inference latency and memory footprint;
- a design space definition - a set of CNN topologies and operands to be explored.

The design flow is configured as a *Selection Procedure* which considers the set of design constraints and the definition of the reference design space to search for the optimal CNN candidate, identified based on several iterations of refinement sub-steps. It is served by a *Dataset management* utility, selecting for the different training actions the pre-processing and data-level transformations to the reference dataset, which defines the CNN-application task. Moreover, the *Selection Procedure* relies on a set of *Evaluation tools* to accurately compare the design points with each other based on the estimation of the most relevant metrics.

In the following, we outline in more detail the features of the tool flow components.

3.1.1 Dataset Management Utility

The ALOHA tool flow enables the customization of the data pre-processing operations according to the application's requirements [97], as showed in the *Dataset management* column of Figure 3.2. In detail, it describes as a plug-in each transformation or pre-processing operator which can be applied to the data, based on the user definition. The plug-ins can be treated independently and arbitrarily connected to others into a *pre-processing pipeline*, which can be applied at sample level, or batch level. For the KWS use case, the *pre-processing plug-ins*

include the feature-extraction functions applied to the audio samples, combined with several *augmentation plug-ins*, like random time shifts, random noise addition, random pitch, and random speed.

3.1.2 Selection Procedure

The *selection procedure* is described in the central column of Figure 3.2. It requires the definition of a reference *Design Space*, and possible design *Constraints*, resulting from the performance requirements of the application, or the resources available on the target hardware.

The constraints definition induces a *Design space pruning* to identify a reduced pool of eligible near-optimal CNN topologies. This step exploits a *Genetic Algorithm* (GA) to surf the search space, ranking and refining the candidate populations of CNNs. As the exploration includes a big number of design points to be compared, their accuracy is assessed at this stage using an efficient one-shot training utility, the *Once-for-All* (OFA) [59], whereas the hardware metrics are estimated based on a target-aware *Latency/ energy/ footprint evaluation* tool.

Finally, a refinement phase is performed on the pruned pool of design points, to precisely assess their accuracy with a *detailed training* and a *quantization* step, reducing their memory footprint: one or more CNN architectures can be selected for *Deployment*.

The OFA training

The one-shot training enables efficient NAS over large design spaces. As anticipated in Section 2.2.3, OFA [59] allows describing the search space as a single SuperNetwork and the design points as all its possible subnetworks. All the models in the design space are thus trained with a single training procedure, requiring a reasonable amount of time, as we will further discuss in Section 3.3.1. The subnetworks are optimized through weight sharing, starting from those having the highest number of parameters, and finally adjusting the accuracy of the smallest ones. This is achieved by considering elastic parameters, such as the kernel size, network's depth, and width. The accuracy of each design point can be assessed by just performing inference over the validation set, which is significantly less time-consuming than repeating an independent training for each design point.

The genetic algorithm

The GA provides a search strategy, selecting populations of design points satisfying the search criteria and updating them. Admittance into the eligible population is evaluated based on accuracy, latency, memory, and energy estimations. The new generations are obtained as the composition of the most accurate points evaluated up to that point, and of new network models randomly obtained with mutations on the parameters of those most promising points. In this work, we consider flexible and independent mutations involving the pre-processing pipeline, the input resolution, the kernel size, the number of convolutional layers, and their width. Thus, the search is guided to the selection of the most suitable candidate points, having the highest validation accuracy within the defined constraints.

3.1.3 Evaluation tools

The right column of Figure 3.2 lists the *Evaluation tools* available for the pruning and refinement phases.

The Latency estimation tool

The *Latency estimation tool* enables hardware-aware performance estimation based on the CNN's parameters and the target platform description. For the NAS targeting SensorTile presented in this Chapter, inference time is evaluated exploiting a simple Roofline-based [79] model, shown in Figure 3.3. The main concepts of the Roofline model are recalled in Section 4.1.3. For the SensorTile, we assessed two distinct performance roofs, representing the maximum achievable performance for convolutional and fully connected operands, set respectively to 0.64 ops/cycle, and 0.3 ops/cycle. This model enables latency estimations having an average 25% error, evaluated on a set of 450 common convolutional layers, and 60 fully connected layers, in comparison to direct measurements.

The tool considers the impact of the different quantization levels on the memory footprint, evaluated assuming a double buffer mechanism for the activations. While the metrics considered in this Chapter do not include energy consumption, a more detailed discussion on the topic of platform-aware performance modeling will be explored in Chapter 4.

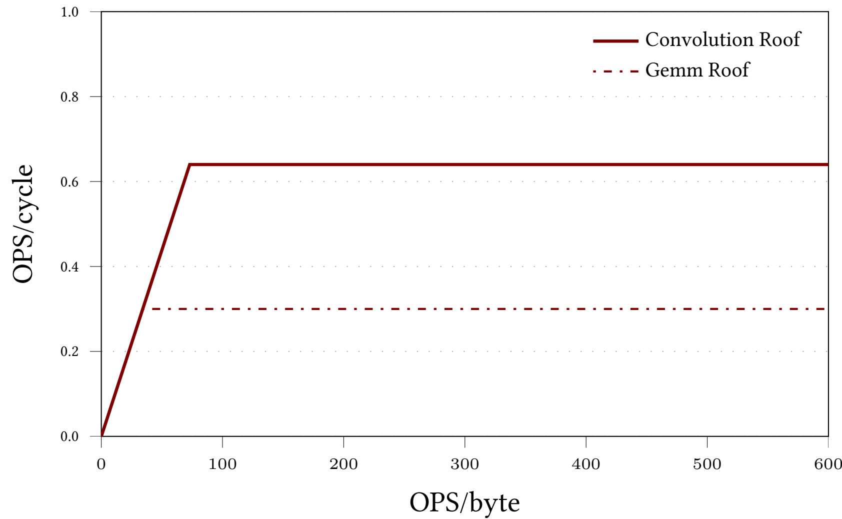


Figure 3.3: Roofline model of the SensorTile platform, representing peak performance roofs for Convolution and Gemm execution.

The Detailed training engine

The *Detailed training engine* allows reaching higher accuracy values than the one-shot training, thanks to the possibility to exploit data augmentation techniques, as supported by the Dataset management plug-ins, although requiring a longer time. It is thus exploited in the refinement phase and applied to a limited set of candidate points. It accepts network models in the ONNX (Open Neural Network Exchange) format [98], and can be exploited to further improve the accuracy of pre-trained models [99].

The quantization engine

The *Quantization engine* is based on the NEMO (NEural Minimization for pytOrch) framework [100], relying on PACT (Parameterized Clipping Activations) quantization [43], whose main concepts were anticipated in Section 2.2.1. To allow quantized representation for the activations and not only the weights, in the resulting network model some operators, such as Batch Normalization (BN) and ReLU, are replaced respectively with a sequence of Mul/Add and Mul/Div/Clip operators.

3.1.4 Deployment

Finally, ALOHA provides a target-oriented *Code generation* tool, to automatically obtain efficient and fast deployment. The tool consists of a Python script generating, from a network model in ONNX format, an intermediate generic representation of NN operators and pre-processing functions, which is finally translated into a C implementation exploiting the specific target-compliant functions, providing the source code for the inference execution as well as the appropriate arrangement of the parameters. In this chapter, we exploited generation targeting SensorTile, based on the CMSIS-NN library [34], slightly modified to efficiently handle the quantized models produced by the quantization tool.

3.2 CNN selection procedure implementation

In the following, we present our proposed efficient design exploration strategy, configured as a CNN selection procedure, based on the ALOHA tools described in Section 3.1. It allows us to treat the network topology, the pre-processing scheme, and the quantization level as random variables during the evolution search. The motivation for the combined feature extraction and topology exploration is further explored in Section 3.3.2. We present two different versions of the selection procedure, detailing the general overview provided in the central column of Figure 3.2. We start from a fast and simple one, whose overview is provided in Figure 3.4a and results in an accurate and efficient, but possibly sub-optimal, selection, obtained with a limited design time. Finally, we describe a more complex and accurate version, shown in Figure 3.4b, relying on a more detailed performance evaluation to improve the selection quality.

3.2.1 Fast implementation

The Fast implementation of our network selection procedure is defined in Algorithm 1. The inputs are represented by a set PP of pre-processing pipelines and a set of hardware constraints, respectively defining the design space of feature extracting functions and spectrum resolutions, and the limits to memory footprint and execution time, according to the platform specifications.

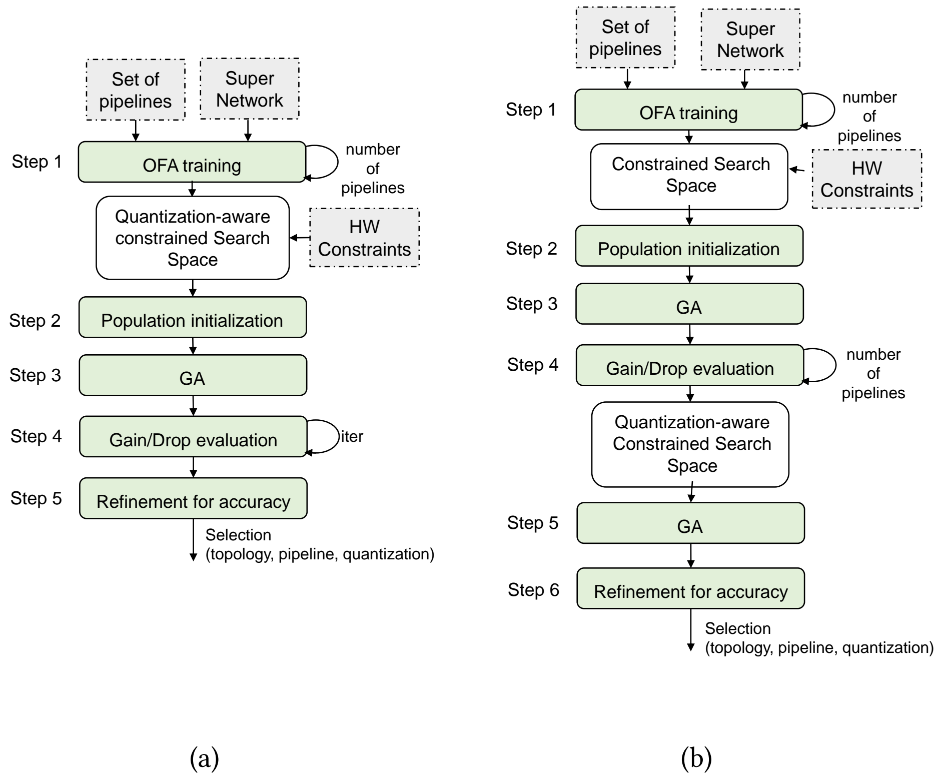


Figure 3.4: Design steps to be performed in the *Design pruning* and *Training refinement* stages (central column of Figure 3.2), according to the a) fast and b) accurate implementations of the CNN selection procedure.

In Step 1, the one-shot training of the SuperNetwork architecture, SN, through the OFA utility [59] defines a design space N of candidate points. A restricted set denoted as $N*_c$ results from the application, in Step 2, of the optional set of constraints: admittance into $N*_c$ is conditioned by the performance evaluation, where quantization up to 4-bit representation is considered for weights and activations independently, to verify the compliance with the maximum allowed memory footprint.

In Step 3, the GA is executed. We consider populations made up of 100 network models, satisfying the hardware constraints, and evolving for G_1=20 generations. The composition of the new generation is obtained as the 25 most accurate network architectures of the previous one, 50 new candidate design points obtained through random mutation of those best-performing design points, and 25 resulting from parameters crossover. The possible mutations involve the network topology, the input resolution, and the feature extraction. The performance evaluation in this step is based on the one-shot training accuracy.

In Step 4 the optimal model is selected: the simplest design choice results in the selection of the most accurate design point in the last generation explored. Alternatively, the one-shot accuracy can be refined with a sequence of evaluation actions, repeated $iter$ times:

- the last generation A_G is ordered based on the predicted classification accuracy (during the first iteration, such value matches the one-shot accuracy);
- the most accurate CNN architecture is selected for 100 epochs of Detailed training, where data augmentation is applied to reduce the overfitting effect;
- the CNN architecture is quantized according to the selected quantization policy, and retrained for 100 epochs to reduce the accuracy drop;
- retraining gain and quantization drop are evaluated and exploited to improve the predicted accuracy of the architectures in A_G.

The user can define the number $iter$ of iterations, which is most suitable to the effort and compute time that he is willing to dedicate to the selection flow. Values of $iter$ different than 1 also require updating the accuracy gain and quantization drop exploited to obtain the predicted accuracy. To provide some examples, gain and drop can be updated by: 1) considering the

Algorithm 1: Fast CNN Architecture selection

Input: $PP(H, W, AudioProc), SN, hw(Mem, Texe)$
Result: CNN architecture a
Step 1. *OFA training*;
for $i \in [1, p]$ **do**
  OFA_train(SN, PP_i);
$N = \{N(PP_1), ..., N(PP_p)\}$;
Step 2. *Population initialization*;
$N*_c = \{n_i|$ (Mem*(n_i),Texe(n_i)) $< hw(Mem, Texe)\}$;
$A_1 = \{n_1, ..., n_{100}\}$ with $n_i \in N*_c$;
Step 3. *GA in HW-aware search space* $N*_c$;
for $i \in [1, G]$ **do**
  Evolution_Search(A_i, $N*_c$);
A_G;
Step 4. *Quantization drop evaluation*;
for $i \in [1, iter]$ **do**
  Order(A_G, Accuracy);
  Detailed_train(best(A_G));
  $g(PP_i)$ = Evaluate_Gain;
  Quantization((n, quant));
  $d(PP_i)$ = Evaluate_Drop;
  Adjust_accuracy(A_G, $g(PP_i)$, $d(PP_i)$);
$a = best(A_G)$;
Step 5. *Refinement for Accuracy*;
return a

values evaluated for the design point with the closest memory footprint; 2) considering an average with the previously evaluated values; 3) considering the last evaluated value.

After the last iteration, Step 5 performs a final refinement on the selected architecture, which can be preceded by the exploration of the optimal learning rate and batch size.

3.2.2 Accurate Implementation

The fast procedure introduces the estimation of the effects of quantization on the model's accuracy only during the final selection process. To remedy this flaw, we also developed a more accurate version, described in Algorithm 2.

In this case, the population initialization in Step 2 defines the constrained search space N_c considering only quantization up to 8 bit, which is the precision targeted by the CMSIS library and typically has very little impact on the network's accuracy. The first run of the GA in Step 3 is exploited as a preliminary step for a more reliable accuracy evaluation, considering refine-

ment with the detailed training and the quantization effect. Following the general assumption that a network with a higher number of parameters can benefit more from the training procedure, we select as the most adequate design points to investigate these effects the CNNs (one for each of the pre-processing pipelines) belonging to A_{G_1} and having an accuracy within one percentage point from the best one, and the biggest footprint: this analysis is exploited as a prediction model for the networks requiring more aggressive quantization to fit the memory constraint and be included in the search.

Therefore, the retraining gain and quantization drop are evaluated in Step 4. In this implementation, the training time on the Detailed engine has a high impact on the overall exploration time. To limit it, we exploit a *static augmentation* of the training dataset, reducing the time dedicated to pre-processing operations and required to obtain different augmentation effects at each epoch: multiple copies of the dataset, enforcing different random levels of data augmentation, are created, saved and made available for successive training procedures. We found that such a solution does not impact the final accuracy.

In Step 5, the GA is executed for a second time, starting from the last generation A_{G_1}, produced in Step 3, and including in the new search space, $N*_c$, the possibility to perform quantization up to 4 bits. At this point, the ranking of the architectures based on their predicted accuracy takes into account the effects evaluated in Step 4.

After G_2=20 generations, the most accurate model, associated with its pre-processing and quantization scheme, is chosen as the optimal selection. Step 6 represents the final refinement phase.

3.2.3 Selection time

The required exploration time for the described selection procedures is quantified in Table 3.1, listing the operations performed according to the fast and accurate implementations, based on measurements performed on NVIDIA Tesla T4, exploited for the one-shot training, and on NVIDIA Tesla P100. The exploration time depends on the search parameters, more specifically on the number of different pre-processing pipelines ($|PP|$) and quantization levels (Q) considered, and on the number of refinement steps performed until selection ($iter$). We report an estimation for each table entry.

Algorithm 2: Accurate CNN Architecture selection

Input: $PP(H, W, AudioProc), SN, hw(Mem, Texe)$
Result: CNN architecture a
Step 1. *OFA train*;
for $i \in [1, p]$ **do**
  OFA_train(SN, PP_i);
$N = \{N(PP_1), ..., N(PP_p)\}$;
Step 2. *Population initialization*;
$N_c = \{n_i|$ (Mem(n_i),Texe(n_i)) $< hw(Mem, Texe)\}$;
$A_1 = \{n_1, ..., n_{100}\}$ with $n_i \in N_c$;
Step 3. *GA in HW-aware search space* N_c;
for $i \in [1, G_1]$ **do**
  Evolution_Search(A_i, N_c);
A_{G_1};
Step 4. *Quantization drop evaluation*;
$D = \{n_{PP_1}, ..., n_{PP_p})\}$ where n_{PP_i} has biggest footprint in A_{G_1} ;
for $n \in D$ **do**
  Detailed_train(n);
  $g(PP_i)$ = Evaluate_Gain;
  Quantization(n);
  $d(PP_i)$ = Evaluate_Drop =$\{d_{x8w8}, d_{x4w8}, d_{x8w4}, d_{x4w4}\}$;
$N{*}_c = \{n_i|$ (Mem*(n_i),Texe(n_i)) $< hw(Mem, Texe)\}$;
$A'_1 = A_{G_1}$;
Step 5. *GA in HW aware Search Space* $N{*}_c$;
for $i \in [1, G_2]$ **do**
  Adjust_accuracy(A'_i, $g(PP)$, $d(PP)$);
  Evolution_Search(A_i, $N{*}_c$);
$a = best(A'_{G_2})$;
Step 6. *Refinement for Accuracy*;
return a

For the use-cases presented in the following, where $|PP| = 6$ and $Q = 4$, the Gain/ Drop evaluation requires 51 hours in the accurate implementation, against 3h 30 needed in the fast one with an $iter$ choice of 1, which does not scale with the number of pipelines and quantization levels explored. Furthermore, we also mean to emphasize the substantial savings deriving from the CNN topology/ pre-processing co-exploration. A separate evaluation would in effect require repeating the topology GA search on multiple design spaces, as many times as is the number of pipelines considered, or assuming in advance a given scheme, neglecting such an important design variable. This would require 36 hours of GA exploration, against the 6 hours needed by the fast implementation. Thus, the fast implementation allows a factor PP reduction of the required exploration time after the one-shot training.

Operation	*Step*		*Execution Time*	
	Accurate Selection	*Fast Selection*	*Accurate Selection*	*Fast Selection*
OFA train	*1*	*1*	*2h 30*$\times$ $\|PP\|$	*2h 30*$\times$ $\|PP\|$
GA	*3*	*3*	*6h*	*6h*
Gain eval	*4*	*4*	*1h 30*$\times$ $\|PP\|$	*1h 30* $\times$ $iter$
Drop eval	*4*	*4*	*2h*$\times$ $Q \times \|PP\|$	*2h* $\times$ $iter$
GA	*5*	-	*6h*	-
Refinement	*6*	*5*	*5h*	*5h*

Table 3.1: Step by step required exploration time for the accurate and fast selection procedure, where the OFA training is executed on NVIDIA Tesla T4, while the GA exploration and the detailed training are executed on NVIDIA Tesla P100.

3.3 Experimental Results

We describe in the following the experimental results deriving from the application of the selection procedure, proposed as thesis contribution, to the design of a KWS system, considering two different deployment scenarios defined based on the state of the art [61, 62], enabling a direct comparison with the literature dealing with NAS for the design of KWS applications.

3.3.1 Search Space definition

The composition of the search space is summarized in Table 3.2. Each CNN design point presents either 1 or 2 convolutional stages, separated by a MaxPooling layer and consisting in 1 to *Max Depth* convolutional layers. Column 3 lists the possible channel width values, while Column 4 summarizes the considered kernel sizes, both set independently for each convolutional layer. The possible feature size within each stage is defined in column 5, while column 6 reports the stage's maximum depth. All the network configurations present a final fully connected stage.

As six pre-processing pipelines are considered, as described in Section 3.3.2, the training process at Step 1 of Algorithms 1 and 2 results in a set of over 330000 CNNs available for exploration, corresponding to all the combination of parameters in Table 3.2 and the pre-

Stage	*Operator*	*Output Features*	*Kernel Size*	*Input Size*	*Max Depth*
0	*Conv*	*16/32/64*	*3x3/5x5*	*40x32/32x16/16x8*	*1*
1	*Conv*	*16/32/64*	*3x3/5x5*	*20x16/16x8/8x4*	*5*

Table 3.2: Parameters of the CNN architectures which constitute the Design Space for NAS targeting SensorTile.

processing schemes.

3.3.2 Preliminary Pre-processing exploration

We started our exploration process by estimating the impact of adapting the pre-processing choice to the hardware target, considering the variability of feature preparation choices reported in the literature. We thus compared different feature-extracting functions and the resulting spectrogram's resolutions. Figure 3.5 shows the output of an evolution search conducted on the search space defined in Section 3.3.1, based on a hardware-aware search strategy evolving by optimizing the design points to be Pareto optimal in terms of classification accuracy and inference time on the target platform, estimated with the *latency evaluation tool* described in Section 3.1. We repeated the search process on six distinct search spaces, each corresponding to a pre-processing scheme choice, exploiting either Mel-spectrogram or MFCC as feature-extracting functions and resulting in 16x8, 32x16, or 40x32 input resolution. Each curve in the figure represents the Pareto fronts obtained after 20 generations, resulting from the different choices of the input resolution and selected feature extracting function. As can be derived from the plot, the overall Pareto front would be made up of design points exploiting different pre-processing schemes: for example, optimal points in the left region are trained on 16x8 Mel-spectrograms, while in the rightmost region of the plot the higher accuracy values are reached thanks to 40x32 spectrograms.

Furthermore, this design choice impacts the overall system performance, as summarized in Table 3.3, reporting the execution time of online pre-processing, measured on the target platform. The measured values do not include the evaluation time of the constant parameters (e.g. the coefficients of the Mel filtering banks, and the DCT matrix), which can be computed

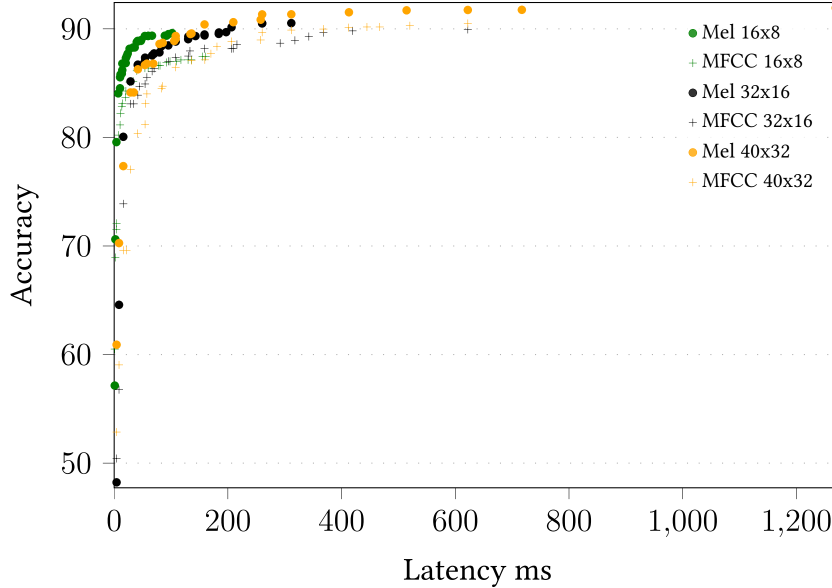


Figure 3.5: Comparison of the Pareto optimal design points resulting from NAS based on distinct evolution searches considering pre-processing schemes based on Mel and MFCC, with input resolutions of 16x8, 32x16, and 40x32.

once at first execution, and memorized for the successive iterations of the audio processing.

Pre-estimating the most suitable pre-processing choice for the target task is thus not trivial. Due to this reason, we consider in the following its co-exploration with the CNN topology and quantization scheme.

3.3.3 Use-case 1

Design Space	***Constraints***	***Constraints Reference***
as defined in Table 3.2	*MOPS: 20* *Memory 200kB*	[61], *referred to as* *Medium region*
	Latency: 390ms	*obtained based on* *model in Figure* 3.3

Table 3.4: Summary of search parameters for NAS targeting use-case 1.

We first considered as a reference the Medium size region defined in [61], as summarized in Table 3.4, considering a maximum memory footprint of 200 kB and a maximum complexity

	Pre-processing time
Mel 16x8	*46 ms*
MFCC 16x8	*48 ms*
Mel 32x16	*94 ms*
MFCC 32x16	*98 ms*
Mel 40x32	*120 ms*
MFCC 40x32	*132 ms*

Table 3.3: Measured execution time for the considered preprocessing schemes on ST SensorTile.

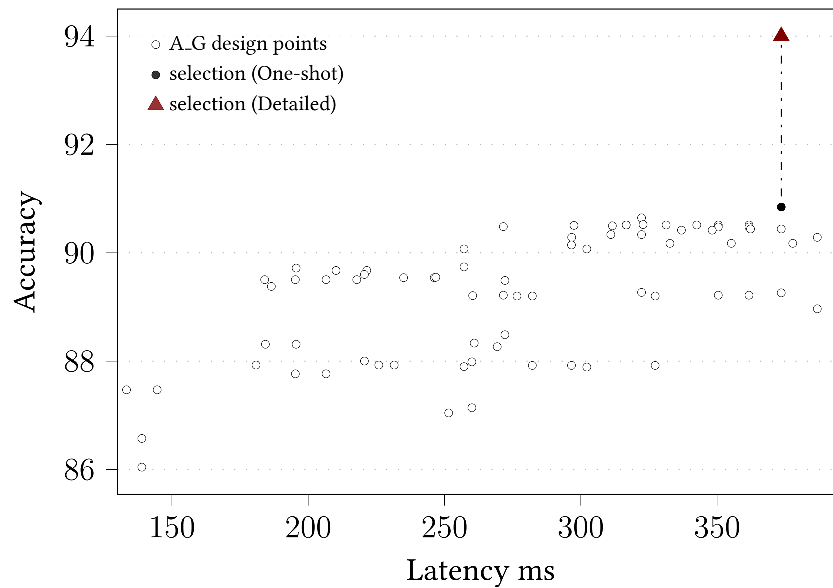


Figure 3.6: Fast selection output in 200kB - 20 MOPS search space. The selected model is highlighted, and its final accuracy upon detailed training is reported.

Network model	***Accuracy***	***Latency***	***MOPS***	***Memory***
CNN M [61]	*92.2%*	*86.9%*	*86.5%*	*99.7%*
fast selection	*94%*	*95.8%*	*86.8%*	*60%*

Table 3.5: Performance metrics summary for the selected design point and the reference state-of-the-art network, in the 200kB-20MOPS region, expressed as a percentage of the constraint's value.

of 20 MOPS, translated into a 390ms latency constraint. The output of the fast selection process on the design space resulting from the constraints applied to the search space defined in Section 3.3.1 is reported in Figure 3.6. Every bullet represents a design point selected by the GA to belong to the last generation, and it is placed based on its estimated latency and its one-shot accuracy. The highlighted point represents the selection resulting from an $iter$ value equal to 1. The selection output also includes the pre-processing and quantization scheme: 8-bit representation for both weights and activations, and Mel-based pre-processing, resulting in 32x16 input spectrograms. The selected network model is finally retrained on the Detailed engine for 100 epochs, exploiting data augmentation through random shifts and random noise addition, and then quantized, resulting in the refined accuracy reported in the plot. Based on our hyper-parameters exploration, the training exploits a learning rate value $lr = 0.025$, batch size $bs = 16$, and SGD optimizer.

The co-exploration approach allows us to improve the efficiency of the design process, since, as shown in Figure 3.5, the pre-processing scheme's impact on performance is deeply connected with the search constraints, and consequently to the CNN architecture to be deployed. Thus, as anticipated in section 3.2.3, performing a dedicated preliminary analysis is not only time-consuming but also very complex, especially when multiple constraints need to be considered. As shown in Figure 3.9, the selected architecture reaches 94% accuracy, improving the state-of-the-art CNN model obtained in [61] by up to 1.8% with 40% lower storage requirements, while the number of OPS is increased by 0.3% and results in a 10% higher latency. The exploration summary is reported in Table 3.5. The search process requires around 30 hours, considering 15 hours of one-shot training executed on NVIDIA Tesla T4 GPU, while the GA and the Detailed training were executed on NVIDIA Tesla P100 GPU.

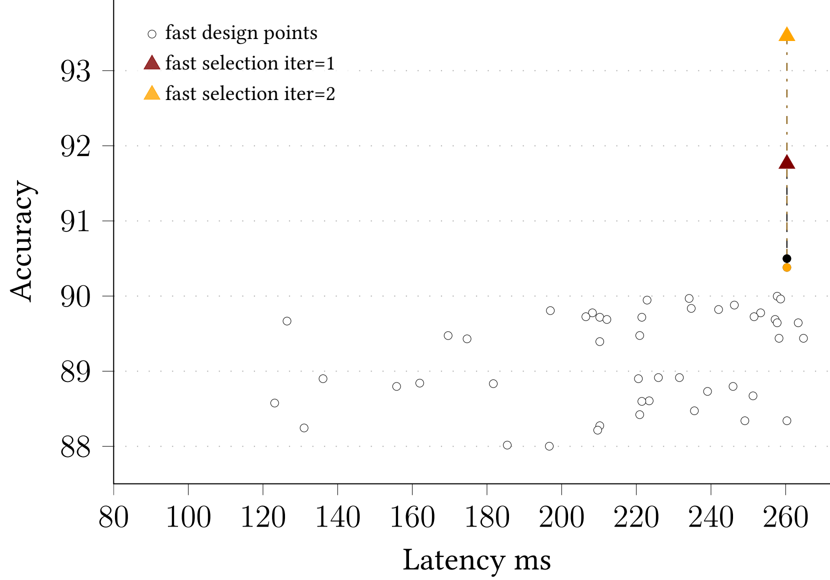


Figure 3.7: CNN architecture fast selection output in 75.7kB - 13.6 MOPS search space. The network models selected based on different choices for the *iter* value are highlighted.

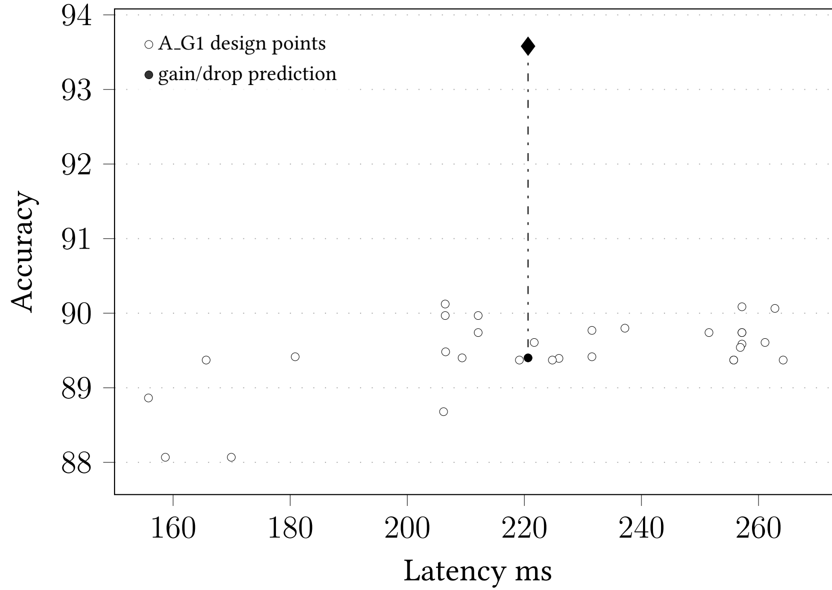
3.3.4 Use-case 2.

Design Space	***Constraints***	***Constraints Reference***
as defined in Table 3.2	*MOPS: 13.6* *Memory 75.7kB*	[62], *parameters of the selected network*
	Latency: 265ms	*obtained based on model in Figure* 3.3

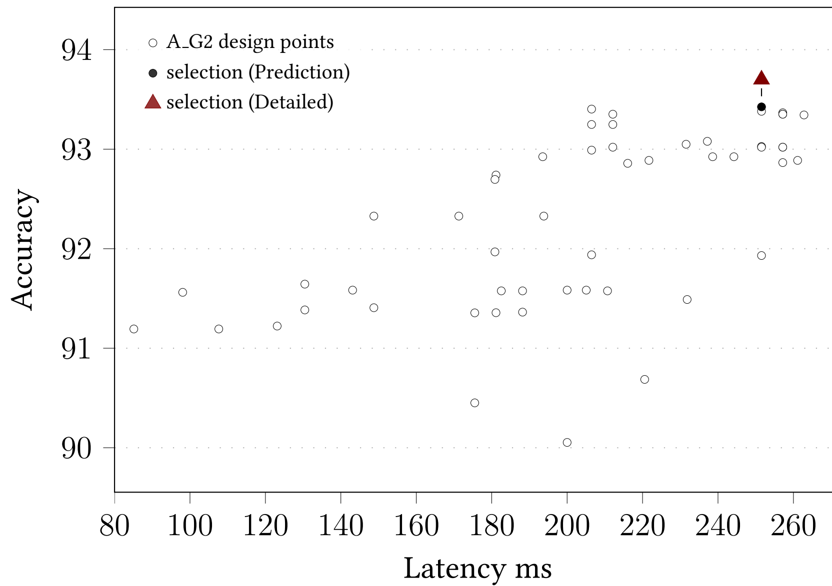
Table 3.6: Summary of search parameters for NAS targeting use-case 2.

As a second use-case, we considered as a reference the work of [62], exploiting the number of OPS and the storage requirements stated for their presented network as the constraints to define the search space. The search parameters are summarized in Table 3.6. The reference CNN model achieves 95.55% accuracy, while its quantized version, exploiting mixed data representation (2.91 bits to represent activations and 2.51 bits to represent weights) reaches 93.76% accuracy.

Figure 3.7 shows the output of the fast selection procedure, reporting two possible selections corresponding to $iter = 1$ and $iter = 2$ values. As can be derived from the plot, the



(a) Pareto plot of the pruned design space after Step 3. The model exploited for the accuracy gain/drop evaluation, on a pipeline based on Mel with 32x16 input resolution, is highlighted.



(b) Pareto plot of the pruned design space after Step 5. The comparison between the predicted and training accuracy on the final selection is highlighted.

Figure 3.8: Accurate selection procedure in 75.7kB - 13.6 MOPS search space.

second one results in a higher accuracy after the refinement process and is referenced as the fast selection in the following. However, considering the accuracy drop connected to the selected quantization level, we applied to this use-case also the accurate implementation of the selection procedure. The corresponding results are described in Figure 3.8. In detail, Figure 3.8a represents the Pareto plot of the most accurate design points belonging to the design space after the preliminary GA run, performed as Step 3 of Algorithm 2, placed according to their one-shot training accuracy, and their estimated latency. We highlight in the plot the design point corresponding to the network model exploited for the quantization drop evaluation, in Step 4, associated with its accuracy projection after the detailed training (performed for 100 epochs exploiting $lr = 0.025$ and $bs = 16$). Although only one design point is depicted in the Figure, corresponding to the pre-processing pipeline based on Mel resulting in 32x16 input resolution, the same gain/drop evaluation is conducted for each of the pipelines considered. The output of the last GA run, corresponding to Step 5, is reported in Figure 3.8b. In this case, the design points are placed according to the accuracy predicted considering the evaluated gain/drop corrections. The plot highlights the resulting selection, having the highest predicted accuracy, and compares it to the one really achieved after the refinement.

Both the fast and accurate procedures select pre-processing based on Mel spectrograms of 32x16 resolution, while they suggest different quantization policies: the fast implementation selects a topology requiring 4-bit representation for weights, whereas the accurate selection can accommodate 8-bit quantization for all datatypes. Figure 3.10 and Table 3.7 report the comparison with the CNN proposed in [62]. The refined accurate selection, after additional 100 refinement training epochs in Step 6, results in an architecture reaching an accuracy 0.14% higher than the one of the quantized version of the reference state-of-the-art architecture, although having higher storage requirements, while neither of the selection procedures allows achieving the accuracy of the full precision model.

Anyway, the fast procedure allows selecting an architecture with an accuracy value of only 0.34% points lower than the accurate one, exploiting only 37% of the required exploration time.

Network model	***Accuracy***	***Latency***	***MOPS***	***Memory***
Full precision [62]	*95.55%*	*100%*	*100%*	*100%*
Quantized [62]	*93.76%*	*100%*	*100%*	*7.8%*
accurate selection	*93.9%*	*94.9%*	*85.9%*	*95.5%*
fast selection	*93.46%*	*98.2%*	*89.7%*	*80.2%*

Table 3.7: Performance metrics summary for the selected design point and the reference state-of-the-art network, in the 75.7kB-13.6MOPS region, expressed as a percentage of the constraint value.

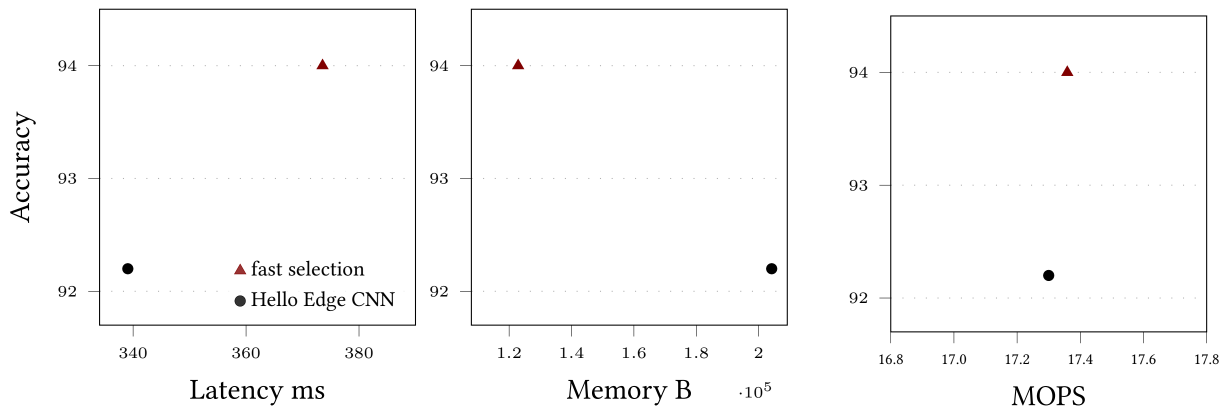


Figure 3.9: Comparison with CNN state of the art in 200kB 20 MOPS region [61].

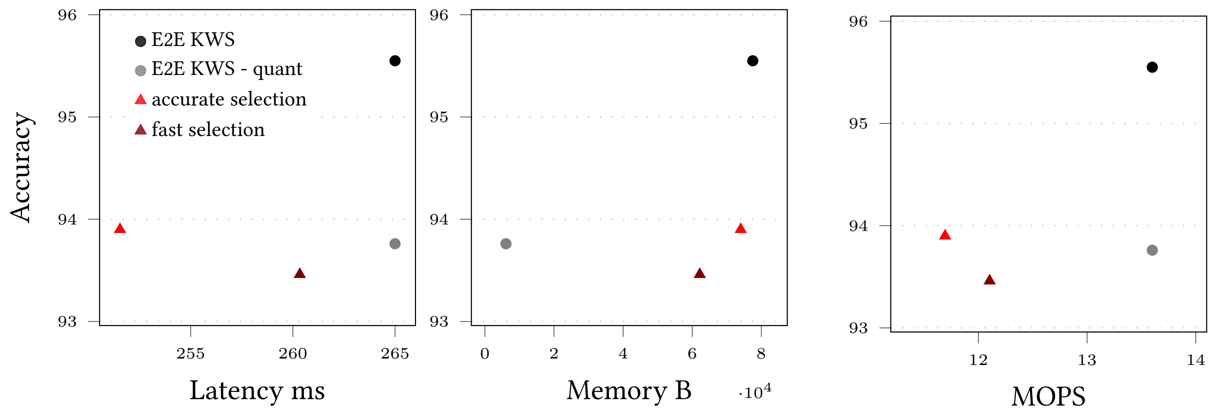


Figure 3.10: Comparison with state-of-the-art End to End KWS network [62].

3.3.5 Selection's quality assessment.

Here we consider use-case 1 to assess the quality of our design flow, evaluating the accuracy of the models selected according to the fast and accurate implementations in comparison with the full exploration of all the design points with detailed training. Figure 3.11 summarizes the outcome of such an extended exploration, showing the accuracy after one-shot training, as well as the predicted and real accuracy after the detailed training exploiting data augmentation, for each of the design points. As can be noticed, the drop/gain prediction provides sufficiently precise results. However, due to some inaccuracy of the one-shot evaluation, both the fast and accurate versions of the design strategy result in the selection of a design point that does not improve as much as it is expected with the detailed training. Since this is not captured by the gain/drop evaluation, the accurate selection requires 38% higher inference time, and has 0.3% lower accuracy than the overall best architecture, highlighted in the plot. However, as shown in Figure 3.12, referring to the fast implementation, a value of $iter = 3$ would be sufficient to find the optimal solution. In this case, the required processing time would slightly increase to 45% of the accurate implementation one, thus still allowing for significant savings.

3.3.6 Closing remarks.

We considered a set of design scenarios for a KWS application, to be deployed on a constrained microcontroller, requiring the evaluation of multiple design parameters resulting in over 330000 CNNs available for exploration. The proposed design procedures allow us to obtain CNN architectures reaching accuracy values competitive with the CNN state-of-the-art in the KWS field while being specifically tailored for the target platform, thanks to hardware-aware inference latency predictions. We obtained up to 1.8% accuracy improvement with 40% lower storage requirements over the best CNN architecture selected in [61] for the 200kB - 20 MOPS Medium region, which exploited 199.4 kB parameters and 17.3 MOPS to reach 92.2% accuracy. In this best-performing use-case, the presented strategy allows to efficiently narrow the design space reaching the final selection in around 30 hours.

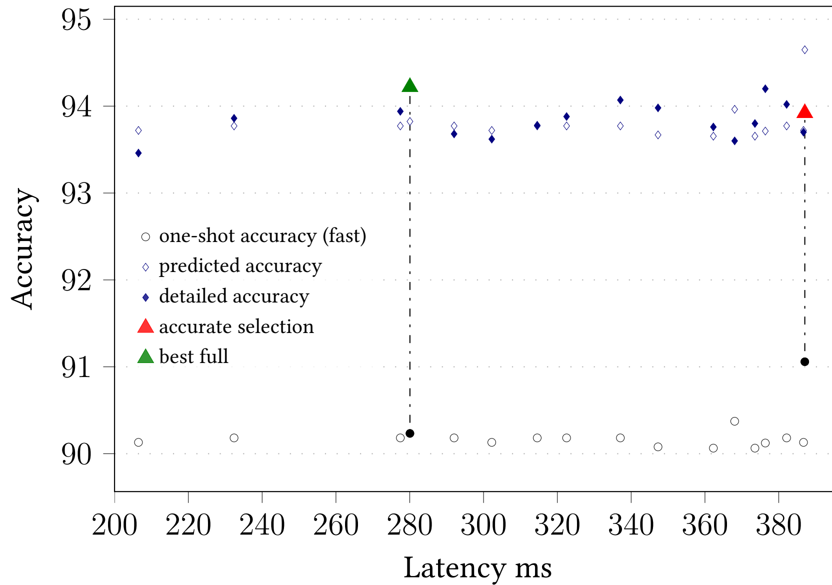


Figure 3.11: CNN architecture accurate selection output in 200kB - 20 MOPS search space. For each design point, the comparison between the predicted accuracy values and the ones achieved after detailed training is reported. The model selected by the accurate procedure is highlighted, as well as the optimal one based on the results of the full exploration.

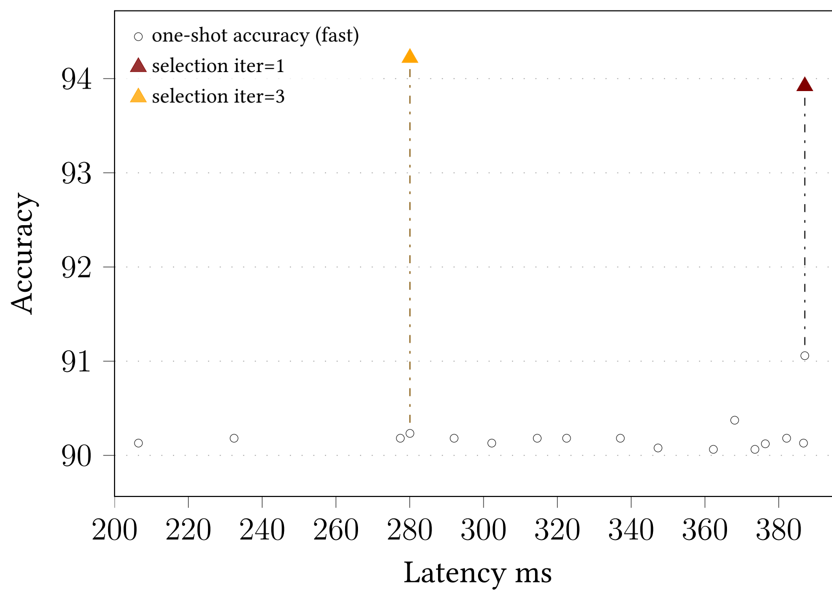


Figure 3.12: CNN architecture fast selection output in 200kB - 20 MOPS search space. The highlighted models represent the final selection resulting from an *iter* value of 1, and of 3, corresponding to the optimal one evaluated through the full exploration.

Chapter 4

Hardware-aware performance estimation: the ALOHA method

Efficient design optimization based on HW-NAS relies on the assumption that an accurate evaluation of the on-hardware performance can be accessed during the search process, thus leading to the selection of a nearly optimal CNN architecture for the task at hand. When more complex heterogeneous and parallel architectures are targeted, simple predictions based on the Roofline model, such as the ones we exploited in the previous chapter, are not sufficiently accurate in modeling the deployment. At the same time, measuring the performance of all the candidate points on the target hardware is often unfeasible and inefficient, given the size of the search space, an issue becoming even more pressing when the system design includes a range of possible hardware targets. The literature reports several works exploiting network design optimization tools based on platform-aware evaluations, but the estimation methods implemented in these tools are inaccurate ([49, 63, 101]), or not sufficiently general ([79, 58, 102, 54, 55, 56, 67, 65]), or their use involves long modeling time and repeated measurements ([103, 81, 70]).

In this chapter, we present our research effort to provide a common unified method to address these issues, implementing platform awareness within automated tools for CNN design. The presented study was conducted in collaboration with the Leiden Institute of Advanced Computer Science (LIACS) and resulted in the definition of a unified method for the evaluation of platform-dependent performance metrics of a CNN deployed on the target hardware.

The presented method is indicated in the following as the ALOHA method, and its definition represents the main contribution described in this chapter. As the main instrument for the estimation, we developed a platform-aware evaluation model, described in Section 4.4 and indicated as ALOHA model, aiming to:

- **provide realistic and accurate results:** the model is capable of capturing platform-aware details, depending on the implemented dataflow and the embedded resources, and impacting the performance, such as the efficiency in exploiting the parallelism enabled by the computing units, or the need to repeat some data transfers during inference execution;
- **be flexible:** the model captures abstract properties, which can describe different platform structures and is not limited to any specific processing element architectural template;
- **be modular:** the model relies on two components, one describing the resources available on the platform, and the other the specific deployment strategy resulting from the selected implementation of CNN layers. As a consequence, different compositions of these two elements can be exploited to accurately model different scenarios;
- **require low development effort:** the information required to define the model can be easily derived from the hardware specifics or a general understanding of the deployment strategy enforced by the library, without involving a benchmarking phase.

We evaluate the accuracy of our proposed method in comparison with the estimations resulting from commonly used alternatives requiring a similar, limited, development effort, showing a significant improvement. We reference two different target platforms, the NEURAghe accelerator, based on a hardware engine implemented on the FPGA and supporting the CPU [32], and the Jetson TX2 module, a system-on-chip (SoC) integrating CPU and GPU [26], which can be considered as representatives of the three main classes of common processing elements in the embedded domain: CPUs, GPUs, and dedicated processing elements. Moreover, we assess the impact of a higher degree of platform awareness on the outcome of a NAS process. Considering the optimal design of a CNN topology for image classification on the CIFAR-10

dataset [45] targeting the reference platforms, we compare the selections resulting from exploiting different hardware-performance estimation methods to evaluate the compliance with a set of user-defined latency constraints. The NAS predictability is significantly improved by the proposed ALOHA method, when compared to the considered abstract and easy-to-use alternatives, resulting in a final design very similar to the one obtained by accessing actual on-hardware measurements during the search process.

To provide a brief outline of the chapter, we recall in Section 4.1 the background concepts and definitions of convolution implementation, and performance estimation based on the OPS count and the Roofline model. After providing a general overview of the proposed ALOHA method in Section 4.2, in Section 4.3 we introduce the ALOHA model for the platform and dataflow description, specifically defined to be exploited by our proposed estimation method, and we provide the example of its implementation for the NEURAGHE accelerator and the Jetson TX2. In Section 4.4 we describe the ALOHA evaluation procedure, exploiting the ALOHA model to provide layer-level hardware metrics estimations, while we present the aggregation module for network-level estimation for sequential and pipelined execution on parallel processors in Section 4.5. Finally, the experimental results are presented in Section 4.6.

4.1 Background

4.1.1 The computational tensor of a CNN layer.

A CNN can be represented as a sequence of layers L, applying a specific functionality to transform the input data X_i into the output data Y_i [73]. Common CNN operators op_i (such as convolution, MaxPooling, GEMM, ReLU etc. [73]) involve processing the input with a sliding window K_i, parameterized with weights W_i. An example of a CNN with layers $L = \{l_1, l_2, l_3\}$ is given in Figure 4.1. The most typical execution schedule involves sequential processing of the CNN layers in $|L|$ computational steps. Given the mismatch between the typical memory footprint of state-of-the-art CNNs and the local storage resources accessed by the computational units of most embedded processing platforms, the layer execution involves three main stages:

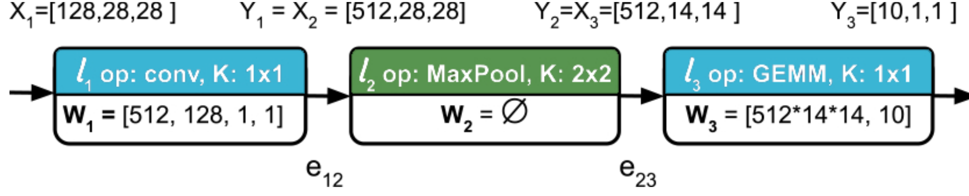


Figure 4.1: CNN

1. the input data X_i and weights W_i are loaded from the global memory into the local memories of the processor allocated for the layer;
2. the workload of the layer is computed on the allocated processor;
3. the computed data Y_i of layer l_i from the local memories of the allocated processor into the platform's global memory;

In this thesis, we represent the data handled during inference execution as tensors T having the format [$T.B$, $T.C$, $T.H$, $T.W$], where $T.B$, $T.C$, $T.H$, $T.W$ represent respectively the batch size, the number of channels, the height, and the width. We particularly focus on the processing case where the batch size is equal to 1, which is the most common for embedded inference execution, as required by most real-time applications. We thus omit the $T.B$ dimension in the notation.

The typical computational workload of a layer can be represented using a set of nested loops enclosing a simple operation, whose size is defined by the dimensions of tensors X_i and Y_i, as well as the sliding window K_i. We thus represent a generic CNN layer as a *computational tensor*, as defined in Listing 4.1. Starting from this general description, the most common CNN layers can be obtained as shown in Table 4.1, replacing the generic loop bounds with the values listed in Columns 2 to 8, and including the operation in Column 9 for the layer in Column 1. The Table can be further extended with new CNN operators. To give an example, we provide in Listing 4.2 the computational tensor representing the first convolutional layer in Figure 4.1: the number of the output and input features of the layer, OF and IF, is replaced with their layer-specific values: $Y_1.C$ = 512, and $X_1.C$ = 128; the height and width of the output features, FH and FW, is equal to $Y_1.H$ = 28, and X_1 = 28, whereas the height and width of the layer sliding window, KH and KW, is replaced by $K_1.H$ = 1, and

CNN op	Computational tensor boundaries							simple op
	BS	**IF**	**OF**	**FW**	**FH**	**KH**	**KW**	
Conv	$X.B$	$X.C$	$Y.C$	$Y.W$	$Y.H$	$K.H$	$K.W$	*MAC*
GEMM	$X.B$	$X.C$	$Y.C$	$X.W$	$X.H$	*1*	*1*	*MAC*
ReLU	$X.B$	*1*	$Y.C$	$Y.W$	$Y.H$	*1*	*1*	*max*
MaxPool	$X.B$	1	$Y.C$	$Y.W$	$Y.H$	$K.H$	$K.W$	*max*

Table 4.1: Layer-specific computational tensor parameter

$K_1.W$ = 1; the generic operation $simple_op$ is configured as a MAC. Finally, the external loop on batch size BS is omitted, assuming the batch size equal to 1.

```
1    for batch in range(BS):
2     for o_feat in range(OF):
3      for i_feat in range(IF):
4       for fh in range(FH):
5        for fw in range(FW):
6         for k_y in range(KH):
7          for k_x in range(KW):
8           do simple_op
```

Listing 4.1: Generic CNN layer computational tensor

```
1    for o_feat in range(512):
2     for i_feat in range(128):
3       for fh in range(28):
4        for fw in range(28):
5         for k_y in range(1):
6          for k_x in range(1):
7           do MAC
```

Listing 4.2: Computational tensor of Convolutional layer l_1

4.1.2 OPS-based Performance prediction

One common approach to latency estimation in the literature refers to the total number of OPS required, as in:

$$t_{OPS} = OPS/AP \tag{4.1}$$

where the attainable performance AP is considered equal to the peak performance of the hardware platform, $APmax$ [OPS/s]. The value of OPS is computed as:

$$OPS = \prod_{n=1}^{N} T.dim_n * \#OPS_enclosed \tag{4.2}$$

where $\prod_{n=1}^{N} T.dim_n$ is the product of all computational tensor dimensions $dim_n, n \in [1, N]$; $\#OPS_enclosed$ is the number of OPS enclosed in the loops of the layer computational tensor. To give an example, for the convolutional layer in Listing 4.2 the total number of operations is evaluated as $512*128*28*28*1*1*2 \approx 102,76*10^6$, where $512*128*28*28*1*1$ is the product of the tensor dimensions and 2 indicates that every iteration of the loop performs two operations: one multiplication and one addition.

4.1.3 Roofline Model

The well-known Roofline Model [79] introduces the impact of memory access on the execution time. It is defined as a curve in the OPS/s vs OPS/byte plane, composed of a horizontal line, representing peak performance, and a diagonal line with 45°inclination, indicating the bandwidth available to off-chip memory. Its use requires the definition of the Operational Intensity, $Int(l_i)$, of a given kernel, obtained as the ratio between OPS count and total data transferred:

$$Int(l_i) = OPS(l_i) / Traffic_{mem}(l_i) \tag{4.3}$$

For a CNN layer l_i the size of data transfers can be estimated as:

$$Traffic_{mem}(l_i) = Size(X_i) + Size(W_i) + Size(Y_i) \tag{4.4}$$

where $Size(X_i)$, $Size(W_i)$ and $Size(Y_i)$ stand for the amount of data (in Bytes) in input data X_i, weights W_i and output data Y_i of layer l_i. The amount of data in a data tensor T is computed as:

$$Size(T) = \prod_{n=1}^{N} T.dim_n * sizeof(pixel) \tag{4.5}$$

where $\prod_{n=1}^{N} T.dim_n$ is the total number of elements in the data tensor; $sizeof(pixel)$ is the number of bytes required to store one element of data tensor T.

The operating point in the curve, which is the intersection with the vertical line representing the kernel's operational intensity, defines the best-case execution time. Thus, the estimation is based on Equation 4.1, with $AP = AP_{roof}$ evaluated as:

$$AP_{roof} = min(AP_{max}, Int * bw) \tag{4.6}$$

where bw is the bandwidth to the off-chip memory.

4.2 ALOHA estimation method

In this section, we describe the workflow of the estimation method representing the second contribution of this thesis, indicated as ALOHA method [1]. Figure 4.2 provides a general overview. The list of expected inputs includes:

- a CNN description (ONNX [98] is an example of accepted format);
- a description of the hardware target based on the ALOHA platform model, introduced in Section 4.3;
- an optional CNN execution configuration, explored in Section 4.5.

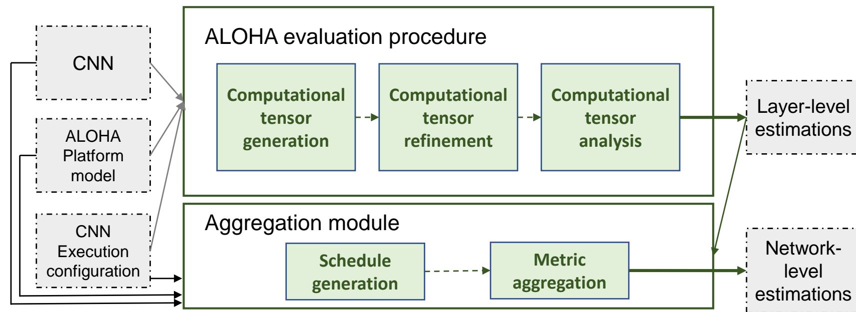


Figure 4.2: Proposed ALOHA estimation method workflow. The required inputs to the different stages are depicted as grey boxes, whereas green boxes represent the single phases of each stage, as described in Sections 4.4 and 4.5.

The definition of the ALOHA description format, as well as the ALOHA evaluation procedure and the aggregation module, represent the different elements of the estimation method, and the main contribution described in this chapter. The estimation process consists of two phases:

- Phase 1: The CNN description and the ALOHA platform model are used by the ALOHA layer-level evaluation procedure, described in Section 4.4 to:

[1] The presented workflow corresponds to the open-source implementation of our ALOHA method, available at https://gitlab.com/aloha.eu/alohaeval

  - generate a computational tensor for every layer in the CNN, based on its parameters; we indicate this process as *Computational tensor generation*, described in Section 4.4.1;
  - refine the computational tensor based on the platform model, setting the correct order of the loops, defining the position of data transfers, mapping the operations on the available processing elements, and partitioning them depending on the limits imposed by storage resources. In this way a new platform-aware computational tensor is obtained for every layer; we refer to this process as *Computational tensor refinement*, described in Section 4.4.2;
  - analyze the refined tensor to derive an accurate estimation of the metrics under evaluation; this process is referred to as *Computational tensor analysis*, described in Section 4.4.3.

- Phase 2: The estimations provided for single layers are exploited as input to an *aggregation* module, described in Section 4.5, delivering the final estimation of the platform-dependent metrics based on the execution schedule.

4.3 ALOHA platform Model

The Roofline model, as well as other simple representations, cannot account for all the details of the platform and the execution data flow, which may have a significant impact on the hardware performance. In order to capture these characteristics, we have defined a more detailed model, which is still abstract and easy to generalize. In this section, we describe the main details, providing two examples in Table 4.2 and Table 4.3, where we apply it to the reference platforms, NEURAghe (see Section 4.3) and Jetson TX2 (see Section 4.3). As long as the vendor provides complete information regarding the working frequency and achievable performance, the local storage size, and the communication resources in the documentation, this description scheme may also be used to model MCU-based platforms. Referring to the example given in Table 4.2 for the description, our platform model is composed of three main elements:

- **Memory resources** (Row 2): lists the available on- and off-chip memory blocks and their size, assigning an ID to each one;
- **IO channels** (Row 3): lists the available connections that can be used to transfer the data between the external storage space and the internal memories, associating the corresponding bandwidth and assigning an ID;
- **Processors** (Rows 4 to 16): lists the set of processors available on the heterogenous platform and sharing the CNN workload. For the sake of brevity, in Table 4.2 and Table 4.3, we provide full processor description only for the platform accelerators.

For every processor in our proposed platform model, we provide two main elements: a *processor description* and a *computational model*, describing how the computational workload is distributed on the resources specified in the platform model. The *processor description* (Rows 4 to 8 and Rows 15 to 16) consists of:

- **General characteristics** (Row 5 and Row 16): it assigns a unique processor identifier (id), and lists the core type and sub-type, top performance, and frequency.
- **Parallelism** (Row 6): it describes the available parallelism as an n-dimensional grid, considering accelerators are commonly implemented as multi-dimensional structures of processing elements. The user must list n parallel factors corresponding to the hierarchy levels organizing the processing element structures in the platform.
- **Power** (Row 7): it provides optional information about the power consumption of the processor, reporting the corresponding value in the active and idle state, and an energy cost per bit accessed in the global memory.
- **Overhead**: (Row 8) it can optionally account for the programming cycles required to start computations on the given processor.

The *computational model* (Rows 9 to 14) defines the following parameters, referring to the generic layer computational tensor, given in Listing 4.1:

- **Loop nesting order and usage** (Row 10): specifies the loops' execution order on the specified processor.

- **Data transfers positioning** (Row 11): specifies the exact positions of data transfers in the computational tensor. As discussed in [104], this parameter can significantly affect the layer latency and energy consumption.

- **IO channels assignment** (Row 12): assigns the IO channels to the transfer input data, output data, and weights.

- **Memory assignment** (Row 13): specifies the assignment of the platform's local memories to input data, output data, and weights. This parameter is used to model the impact of limited memory resources on the layer execution, causing certain loops to be *tiled*, partitioned in portions that can be handled with the data fitting those memories. To specify which of the loops in the computational tensor is affected, we assign it the limiting memory identifier.

- **Parallelism levels assignment** (Row 14): describes how the different degrees of parallelism available are used to partially unroll the convolution loops, associating a computational tensor loop level with each dimension of the parallel computational grid of the platform accelerator.

Example 1: describing NEURAghe

NEURAghe is a convolution accelerator that can be implemented with different parameterization, but we consider in this thesis a setup deployed on Ultra96 board by Avnet, embedding a Xilinx Zynq UltraScale+ MPSoC ZU3EG A484 and a RAM Micron 2 GB LPDDR4 Memory. The memory subsystem in NEURAghe includes four storage spaces, defined in the *Memory Resources* slot of Table 4.2. The memories specified as *memory 0, memory 1* and *memory 2* are local to the hardware convolution accelerator available on the platform and are respectively destined to weights, activation data, and computed results, while the last one, specified as *memory 3* is the off-chip memory, shared between the hardware accelerator and the general-purpose processor. The data transfers between the global and local memories are handled through three separate DMA channels, transferring 8 B/cycle, described in Table 4.2 as *IO channel 0* and *1*, operating at 90 MHz, and *IO channel 2* operating at 180 MHz.

NEURAGHE Platform			
Memory resources	*ID*	*Size*	
	0	73728 B	
	1	163840 B	
	2	92160 B	
	3	2097152 B	
IO channels	*ID*	*Bandwidth (BW)*	
	0	0.72 GB/s	
	1	0.72 GB/s	
	2	2.88 GB/s	
Processor description			
General characteristics	*id*	0	
	type/sub-type	accelerator/FPGA engine	
	top performance	129.6 GOPs/s	
	frequency	0.18 GHz	
Parallelism	*Level*	*Dimension*	*Description*
	level 0	9	MAC matrix has
	level 1	10	9*10 MACs,
	level 2	4	4 pixels/cycle
Power	*active power*	3.6W	
	idle power	1.8W	
	bit access to DDR	91pJ	
Overhead	0.1 ms		
Computational Model			
Loop nesting order and usage	*Loop iterating on*	*Assigned order*	
	OF	level 1	
	IF	level 0	
	FH	level 2	
	FW	level 3	
	KH	level 4	
	KW	level 5	
Data transfers positioning	*Transfer type*	*at loop level*	
	Input Features	level 1	
	Output Features	level 0	
	Weights	level 1	
IO channel assignment	*Transfer type*	*to channel ID*	
	Input Features	0	
	Output Features	1	
	Weights	2	
Memory assignment	*Data type*	*to memory ID*	*limited loop*
	Input Features	0	FH (Loop level 2)
	Output Features	1	OF (Loop level 1)
	Weights	2	OF (Loop level 1)
Parallelism levels assignment	*Level*	*to loop iterator*	
	0	IF (Loop level 0)	
	1	OF (Loop level 1)	
	2	FW (Loop level 3)	
Processor description			
General characteristics	*id*	1	
	type/sub-type	CPU/Arm Cortex-A53	
	top performance	9,6 GOPs/s	
	frequency	1,2 GHz	
...			

Table 4.2: ALOHA platform model for NEURAGHE

<table>
<tr><td colspan="4">Jetson Platform</td></tr>
<tr><td rowspan="4">Memory resources</td><td>ID</td><td colspan="2">Size</td></tr>
<tr><td>0</td><td colspan="2">8589934592 B</td></tr>
<tr><td>1</td><td colspan="2">131072 B</td></tr>
<tr><td>2</td><td colspan="2">524288 B</td></tr>
<tr><td rowspan="4">IO channels</td><td>ID</td><td colspan="2">Bandwidth (BW)</td></tr>
<tr><td>0</td><td colspan="2">20 GB/s</td></tr>
<tr><td>1</td><td colspan="2">20 GB/s</td></tr>
<tr><td>2</td><td colspan="2">35 GB/s</td></tr>
<tr><td colspan="4">Processor description</td></tr>
<tr><td rowspan="4">General characteristics</td><td>id</td><td colspan="2">0</td></tr>
<tr><td>type/sub-type</td><td colspan="2">accelerator/GPU</td></tr>
<tr><td>top performance</td><td colspan="2">666.6 GOPs/s</td></tr>
<tr><td>frequency</td><td colspan="2">1.3 GHz</td></tr>
<tr><td rowspan="4">Parallelism</td><td>Level</td><td>Dimension</td><td>Description</td></tr>
<tr><td>level 0</td><td>2</td><td>MAC matrix contains</td></tr>
<tr><td>level 1</td><td>16</td><td>x 2 SM x 16 blocks</td></tr>
<tr><td>level 2</td><td>128</td><td>per SM x 128 cores</td></tr>
<tr><td>Power</td><td>active power</td><td colspan="2">15W</td></tr>
<tr><td>Overhead</td><td colspan="3">0.01 ms</td></tr>
<tr><td colspan="4">Computational Model</td></tr>
<tr><td rowspan="7">Loop nesting order and usage</td><td>Loop iterating on</td><td colspan="2">Assigned order</td></tr>
<tr><td>OF</td><td colspan="2">level 0</td></tr>
<tr><td>IF</td><td colspan="2">level 1</td></tr>
<tr><td>FH</td><td colspan="2">level 2</td></tr>
<tr><td>FW</td><td colspan="2">level 3</td></tr>
<tr><td>KH</td><td colspan="2">level 4</td></tr>
<tr><td>KW</td><td colspan="2">level 5</td></tr>
<tr><td rowspan="4">Data transfers positioning</td><td>Transfer type</td><td colspan="2">at loop level</td></tr>
<tr><td>Input Features</td><td colspan="2">level 0</td></tr>
<tr><td>Output Features</td><td colspan="2">level 0</td></tr>
<tr><td>Weights</td><td colspan="2">level 0</td></tr>
<tr><td rowspan="4">IO channel assignment</td><td>Transfer type</td><td colspan="2">to channel ID</td></tr>
<tr><td>Input Features</td><td colspan="2">1</td></tr>
<tr><td>Output Features</td><td colspan="2">0</td></tr>
<tr><td>Weights</td><td colspan="2">0</td></tr>
<tr><td rowspan="4">Memory assignment</td><td>Data type</td><td>to memory ID</td><td>limited loop</td></tr>
<tr><td>Input Features</td><td>1</td><td>OF (Loop level 0)</td></tr>
<tr><td>Output Features</td><td>0</td><td>OF (Loop level 0)</td></tr>
<tr><td>Weights</td><td>0</td><td>OF (Loop level 0)</td></tr>
<tr><td rowspan="4">Parallelism levels assignment</td><td>Level</td><td colspan="2">to loop iterator</td></tr>
<tr><td>0</td><td colspan="2">IF (Loop level 1)</td></tr>
<tr><td>1</td><td colspan="2">OF (Loop level 0)</td></tr>
<tr><td>2</td><td colspan="2">FH, FW (Loop level 2, 3)</td></tr>
<tr><td colspan="4">Processor description</td></tr>
<tr><td rowspan="4">General characteristics</td><td>id</td><td colspan="2">1</td></tr>
<tr><td>type/sub-type</td><td colspan="2">CPU/ARM Cortex A-57</td></tr>
<tr><td>top performance</td><td colspan="2">16.28 GOPs/s</td></tr>
<tr><td>frequency</td><td colspan="2">2.35 GHz</td></tr>
<tr><td colspan="4">...</td></tr>
</table>

Table 4.3: ALOHA platform model for Jetson TX2

The computational resources consist of an ARM Cortex-a53 core exploited as a general-purpose processor and a convolution-specific FPGA-based accelerator. Table 4.2 only describes the hardware accelerator.

The considered configuration features a matrix of 90 MAC modules, distributed over 9 parallel input channels and 10 parallel output channels, working at 180 MHz clock frequency. Moreover, each MAC module in NEURAGHE is designed to process four neighboring pixels in an input row per cycle. Table 4.2 models its computing resources by defining, in the *Parallelism* field, a $level0$ and $level1$ parallelism, respectively set to 9 and 10 and representing the dimensions of the computational grid, and a $level2$ parallelism, set to 4 and corresponding to the number of pixels processed per cycle. Thus the platform is able to deliver a peak performance of 129,6 GOPS/s for 16-bit CNN data precision. The platform power consumption was assessed using the Xilinx Power Estimator tool [105], obtaining P_{act}=3,6 W for the active state, and P_{idle}=1,8 W for the idle state. Moreover, we have accounted for DDR energy consumption. To this aim, we have used the DRAMpower tool [106], fed with transaction traces obtained by RTL simulation. We obtained a per-bit energy contribution of En_{bit}=91 pJ/bit for a 4Gb Micron LPDDR3 memory. A typical CNN execution data flow on the platform is described in the *Computational model* field. Parallelism levels are linked to their corresponding loop levels in the *Parallelism level assignment section*, by referring to the specific nesting order implemented in the platform, and defined in the *Loop nesting order and usage* section. The $level0$ parallelism is exploited to unroll computations over IFs, while the $level1$ parallelism, defines unrolling over OFs and $level2$ parallelism allows unroll by a factor of 4 the X loop. The *Memory assignment section* defines how CNN data is stored in each of the storage spaces available, and how their limited size affects the execution dataflow of a CNN layer.

Example 2: describing Jetson

Jetson TX2 [26] is a GPU-based platform from NVIDIA. The memory system includes three defined storage spaces, listed in Table 4.3 as *memory 0*, *memory 1*, and *memory 2*, indicating a shared 1.866-GHz DRAM memory, directly accessed by all platform processors, a local GPU memory of total size 128 KB, and a shared L2 cache with a configurable size of 512 KB to 2 MB. Separate data transfer channels, reported in Table 4.3 as *IO channels 0, 1, 2* are available

for communications between the global memory and other platform memories. The computational resources of the Jetson TX2 platform are constituted by an NVIDIA Pascal GPU, a quad-core Dual-Core NVIDIA Denver 2 64-Bit CPU, and a quad-Core ARM Cortex-A57 MPCore. For brevity reasons, we only provide in Table 4.3 the full description of the platform GPU. The GPU processor of the NVIDIA Jetson TX2 has two Streaming Multiprocessors (SMs), each having 128 1.3-GHz cores and capable of running 2048 threads, organized in 2048/128=16 thread blocks. These resources are described in the *Parallelism* field of Table 4.3 as three parallelism levels: e͡xtitlevel 0 having size 2 and referring to the number of SMs, a *level 1* having size 16 and referring to the number of blocks available for each SM, and a *level 2* having size 128 and referring to the number of threads executed per block. The GPU reaches a peak performance of 666.6 GOPs/s for FP32 CNN data precision (see field *General characteristics* in Table 4.3). CNN inference is typically performed using the TensorRT DL framework [107], provided by NVIDIAs as an official DL framework for the platform. We refer to the TensorRT implementation to describe how the parallelism is exploited and report the information in the *Loop nesting order and usage* field of Table 4.3. As specified in the *Memory assignment* field, the output data and the weights of a CNN layer are stored in the global platform memory, while the input data is stored in the shared GPU memory. The sizes of the platform memories affects the execution of the CNN workload as specified in the *Memory assignment* field of Table 4.3.

4.4 ALOHA evaluation Procedure

We describe at this point how our proposed ALOHA evaluation procedure is applied, to obtain a fast, yet accurate, target-oriented evaluation of the CNN layer's performance. The increased reliability over the Roofline model and the OPS-based estimations is obtained in our contribution by accounting for:

- **repeated transfers** of the layer input data and weights to the local memories of the platform processors, occurring when the size of the data to be stored exceeds the size of the local storage space. The same effect can be observed for the output data transferred from the local memory to the global one, causing additional time and energy overheads during the CNN layer execution;

- **occupancy/rounding effect**, representing a waste of the available computational power, caused by the inefficient exploitation of the parallel computing units on the platform. It is typically measured in terms of wasted computational cycles, or partial processor occupancy, and translated in reduced performance of the platform computational resources [104];
- **separate bandwidth ceilings** instead of considering only the peak memory bandwidth, which accounts for high utilization of all data communication channels, it includes the communication overheads caused by an uneven distribution of the CNN layer data (input data, output data, and weights) over the platform memories and the data communication channels.

As anticipated in 4.2, the evaluation procedure involves three main phases:

- *Computational tensor generation.* This phase generates a representation of a CNN layer enabling explicit specification of the parallelism available within the CNN layer.
- *Computational tensor refinement.* In this phase, the generated platform-agnostic layer computational tensor is refined of platform-aware parameters derived from the proposed ALOHA platform model, to obtain a platform-aware computational tensor.
- *Computational tensor analysis.* This phase involves the final estimation of the platform-dependent metric of interest from the refined computational tensor.

A detailed description of each phase is given in Section 4.4.1, 4.4.2, and 4.4.3, using as an example the convolutional layer l_1, shown in Figure 4.1, executed on the NEURAGHE platform, modeled with our proposed ALOHA platform description in Table 4.2.

4.4.1 Computational tensor generation

Starting from the generic CNN layer representation as a 6-dimensional computational tensor, given in Listing 4.1, our proposed ALOHA evaluation procedure exploits Table 4.1 to generate the layer computational tensor. For example, for the CNN layer l_1, shown in Figure 4.1, the ALOHA procedure generates the CNN layer computational tensor provided in Listing 4.2.

4.4.2 Computational tensor refinement

At this point, specific platform-aware transformations are applied to the generated computational tensor, in four steps (see Steps 1 to 4 below). We show as an example how the platform-agnostic computational tensor in Listing 4.2 is refined for modeling the execution on the NEURAGHE platform.

- **Step 1**: Apply **loop nesting order and usage** to the computational tensor loops. This step swaps lines 1 and 2 in Listing 4.2, resulting in Listing 4.3;

- **Step 2**: Apply the unrolling defined by the **parallelism level assignment** to the corresponding computational tensor loop. During this step, an indented loop par_*, representing parallel computations, is inserted in the nested structure, according to the *Parallelism level assignment* field. As a consequence, the number of iterations of the new pair of loops is rounded over the corresponding computational grid dimension. For example, the level 0 parallelism of size 9, shown in Table 4.2, and assigned to the IF loop of the computational tensor, causes the insertion of loop par_0 with 9 iterations in Listing 4.4 (line 2), and rounding of the IF loop (line 1) to roundup(128/9) = 15 iterations. Analogously, the level 1 parallelism of size 10, shown in Table 4.2, and assigned to the OF loop of the computational tensor, causes the insertion of loop par_1 with 10 iterations in Listing 4.4 (line 4), and rounding of loop OF (line 3) to roundup(512/10) = 52 iterations.

- **Step 3**: Introduce **data transfers**, i.e., explicitly specify when the transfer of the layer input data, output data, and weights is performed in the layer computational tensor. Every data transfer is assigned to a specific loop, as described in the *data transfer positioning* field of the platform computational model, and is represented as a line $action(data_bytes, mem_i, ch_j)$, where $action \in (load, store)$ specifies the direction of the transfer. If $action = load$, the data transfer is placed before the computations within the assigned loop are performed. If $action = store$, the data transfer is placed after the computations within the assigned loop are performed; $data_bytes$ specifies the amount of data (in bytes) transferred during the data transfer action and is assessed for every op/data type, using specific properties of the CNN layer and the layer computational

tensor. The parameter mem_i specifies the platform memory where data is accumulated, whereas ch_j specifies the IO channel used. For example, in Listing 4.5, this step leads to the insertion of line 2, where the input data of size $9 * 28 * 7 * 4 * 2$ bytes is loaded from the device main memory into the processor local memory mem_0 through the data communication channel ch_0. How to evaluate the data transfer size is further detailed in Equation 4.7, introduced in the following phase, describing the *Computational tensor analysis*.

- **Step 4**: Pose **memory constraints** to the size of computational tensor loops. During this step, the evaluation procedure checks the utilization of the platform memory within the loops associated with a limited platform memory, as specified in the *Memory assignment* field of the platform model. If the memory constraint is violated, the loop is tiled. For example, as specified in Table 4.2, the OF loop of the computational tensor is limited by the local memory mem_1 of size 163840 bytes. In Listing 4.5, the layer tries to accumulate 815360 bytes in memory mem_1 (line 13), thus violating the constraint placed by memory mem_1 on the OF loop, as $815360 > 163840$. This causes the introduction of the additional loop out_tile (line 1 in Listing 4.6) with 6 iterations, and the reduction of the OF loop (line 6 in Listing 4.6) bounds to 52/6 = 9 iterations. In Listing 4.6, the layer stores $9 * 10 * 28 * 7 * 4 * 2$ = 141120 bytes < 163840 bytes of output data in memory mem_1 at each iteration of loop out_tile, thus satisfying the memory constraint.

```
1  for i_feat in range(128):
2   for o_feat in range(512):
3    for fh in range(28):
4     for fw in range(28):
5      for k_y in range(1):
6       for k_x in range(1):
7        do MAC
```

Listing 4.3: Step 1. Loops nesting order and usage

```
1  for i_feat in range(15):
2   for par0 in range(9):
3    for o_feat in range(52):
4     for par1 in range(10):
5      for fh in range(28):
6       for fw in range(7):
7        for par2 in range(4):
8         for k_y in range(1):
```

```
9            for k_x in range(1):
10            do MAC
```

Listing 4.4: Step 2. Parallelism levels assignment

```
1  for i_feat in range(15):
2   load(9*28*7*4*2, mem0, ch0) #input data
3   load(9*52*10(1*1+1)*2, mem2, ch2) # weights
4   for par0 in range(9):
5    for o_feat in range(52):
6     for par1 in range(10):
7      for fh in range(28):
8       for fw in range(7):
9        for par2 in range(4):
10        for k_y in range(1):
11         for k_x in range(1):
12          do MAC
13 store(52*10*28*7*4*2, mem1, ch1) #output data
```

Listing 4.5: Step 3. Data transfers introduction

```
1  for out_tile in range(6):
2   for i_feat in range(15):
3    load(9*28*7*4*2, mem0, ch0) #input data
4    load(9*9*10(1*1+1)*2, mem2, ch2) # weights
5    for par0 in range(9):
6     for o_feat in range(9):
7      for par1 in range(10):
8       for fh in range(28):
9        for fw in range(7):
10        for par2 in range(4):
11         for k_y in range(1):
12          for k_x in range(1):
13           do MAC
14  store(9*10*28*7*4*2, mem1, ch1) #output data
```

Listing 4.6: Step 4. Limits posing (tiling)

4.4.3 Computational tensor analysis

In this phase, our proposed ALOHA evaluation procedure exploits the platform-aware computational tensor, to accurately quantify the total number of operations and data transfers performed during the execution of the CNN layer. According to Equation 4.2, the total number of operations OPS_{re} performed by the refined computational tensor in Listing 4.6, is computed as OPS_{re}[2]$= 6 * 15 * 9 * 9 * 10 * 28 * 7 * 4 * 1 * 1 * 2 \approx 110,07 * 10^6$.

[2]Description in Listing 4.6 is simplified for the reader. It does not present some details, e.g. in the last iteration of the loop at line 2, the loop at line 6 stops as soon as roundupOF = 520 OFs have been processed.

We note that this number of operations does not match the total number of operations $OPS_{th} = 102,76 * 10^6$, computed in Section 4.1.2 considering the platform-agnostic CNN layer computational tensor in Listing 4.2. The difference between OPS_{re} and OPS_{th} estimations results from an imperfect distribution of the layer computations over the platform processors, thus some of the computational cycles in the implementation in Listing 4.6 are wasted. We refer to this as the *rounding effect*. The refinement of the layer computational tensor with platform details enables for consideration of the rounding effect and therefore enables for more precise representation of the CNN layer execution on the target platform.

Analogously, the actual amount of memory transfers, impacting the layer's operational intensity, is assessed. Equation 4.5 defines the theoretical transfers based on data tensor dimensions, under the assumption that all of the data can be transferred at once to local memories and made available throughout the entire computation. However, our proposed ALOHA method considers how the specific nesting structure implemented impacts the actual memory traffic.

The amount of data transferred for every data tensor can be evaluated as:

$$Size_{re}(T) = \prod_{n=1}^{N} T.dim'_n * sizeof(pixel) * iterations_{tl} \tag{4.7}$$

where, if one of the $T.dim_n$ dimensions of data tensor T is subject to partitioning among multiple loops, we define as $T.dim'_n$ the dimension that is handled in the convolution loops internal to the transfer level tl, which is subject to a certain number of $iterations_{tl}$, based on the loop nesting structure.

Finally, given the IO channel assignment, the ALOHA method evaluates the operational intensity over single available channels, exploiting Equation 4.3, where the OPS count is evaluated in detail, considering rounding effects and tiling according to Listing 4.6, and the traffic value accounts for repeated transfers, based on Equation 4.7.
At this point, it is possible to use an approach inspired by the Roofline model, but turning Equation 4.6 into:

$$AP_{ALOHA} = min(AP_{max}, Int_{ch0} * bw_{ch0}, ..., Int_{chn} * bw_{chn}) \tag{4.8}$$

Considering $AP = AP_{ALOHA}$ in Equation 4.1, the execution time is evaluated as:

$$t_{ALOHA} = OPS_{re}/(AP_{ALOHA}) + ov \tag{4.9}$$

where a known programming overhead is added to the predicted value as a fixed offset ov.

4.5 CNN metric aggregation

We describe in this section our CNN metric aggregation module. As anticipated in Section 4.2, this module accepts as inputs the estimations of the hardware metrics of single CNN layers and aggregates them to deliver the network-level estimations, in terms of latency, energy consumption, and throughput.

Some research efforts have demonstrated that systems with multiple accelerators can be useful also in the embedded domain [108, 109], although not very common. The ALOHA method introduced in this chapter thus allows taking into account an arbitrary number of processing elements. We neglect any possible contention on the off-chip memory since previous experiments have shown that the effect of this issue is limited, as shown in [108] for multiple instances of NEURAghe accessing the same DDR memory. We also assume asynchronous communication with the host, thus we do not consider the host CPU intervention to become a bottleneck, aligning with most of the approaches in the literature.

As discussed in Section 4.2, the aggregation module accepts an optional *CNN execution configuration* input, describing:

1. the distribution of CNN operators over the target platform processors;

2. whether task-level (pipeline) parallelism is exploited.

The first one specifies how the CNN layers should be distributed over the heterogeneous processors in a target platform, which can be dedicated to different sets of operators (considering NEURAghe as an example, only some of the CNN operators can be accelerated on the FPGA, and the rest of the CNN operators are performed on the platform CPUs). Formally, we define the assignment as a set of tuples $op_dist = \{(op, proc_type)\}$, where op is a CNN operator (such as Convolution or Pooling) and $proc_type$ is the type of a platform processor

(e.g. CPU or GPU). For example, a set of tuples $\{(conv, accelerator), (gemm, CPU)\}$ specifies that convolutions are always offloaded to the platform accelerator, whereas GEMM are always performed on the platform CPUs. If no processor type is specified for a given operator, the aggregation module assumes that the corresponding layer can be executed on any available processor.

The second one specifies if a CNN is executed sequentially or as a pipeline. Sequential execution allows processing only one of the CNN layers at every moment in time, and it is the typical schedule for the majority of widely used DL frameworks such as PyTorch [71] or TensorFlow [72]. On the other hand, when the execution is pipelined, several CNN layers can be executed in parallel to process a different input, resulting in a higher throughput [74, 75]. The execution schedule should thus be taken into account when delivering the system-level performance evaluations. We formalized it as a flag $pipeline$ set to $true$ or $false$ (default).

Based on these inputs, a CNN schedule generator generates an execution schedule J for the evaluated network, assigning each layer $l_i \in L$ a starting time $s_i \geq 0$ and a processor $PE_j, j \in [1, M]$ to be executed on. The supported schedules are $sequential$ or $pipeline$, although the tool can be extended with additional options.

Algorithm 3 shows how a sequential schedule is generated: it starts in Line 1 from an empty schedule J and sets the current starting time s to 0. In Lines 2 to 17, a starting time $s_i \geq 0$ and a processor $PE_j \in PE$ are assigned to every layer $l_i \in L$. In Lines 3 to 9, a list of processors $PE_{suitable}$ suitable for the execution of layer l_i is created, based on the tuples specified in the distribution op_dist. In Lines 10 to 14, Algorithm 3 selects, from the list of suitable processors $PE_{suitable}$, the processor PE_j providing the shortest execution latency $t(l_i, PE_j)$. In Lines 15 to 17, Algorithm 3 assigns time $s_i = s$ and processor PE_j to the layer l_i (Line 15) and updates the starting time s by adding the estimated latency. Finally, in Line 18, Algorithm 3 returns the sequential schedule of the input CNN.

As for the pipeline schedule, it is generated according to the heuristic algorithm proposed in [74]. At this point, the CNN metric aggregation sub-module uses Equation 4.10, Equation 4.11, and Equation 4.12 to estimate CNN latency t_{CNN} (in seconds), CNN throughput Th_{CNN} (in frames per second) and CNN energy cost En_{CNN} (in Joules), respectively.

Algorithm 3: Sequential schedule generation

```
Input: CNN(L, E), PE, op_dist, {t(l_i, PE_j)}
Result: CNN schedule J
J = ∅; s = 0;
for i ∈ [1, |L|] do
    PE_suitable = ∅;
    if ∃(op, proc_type) ∈ op_dist : op = l_i.op then
        for (l_i.op, proc_type) ∈ op_dist do
            for PE_j ∈ PE : PE_j.type = proc_type do
                PE_suitable = PE_suitable + PE_j;
    else
        PE_suitable = PE;
    PE_j = PE_suitable.pop();
    while PE_suitable ≠ ∅ do
        PE_k = PE_suitable.pop();
        if t(l_i, PE_k) < t(l_i, PE_j) then
            PE_j = PE_k;
    s_i = s;
    J = J + (s_i, PE_j);
    s = s + t(l_i, PE_j);
return J
```

$$t_{CNN} = s_{|L|} + t(l_{|L|}) - s_1 \tag{4.10}$$

$$Th_{CNN} = \begin{cases} 1/max_j \sum_{(s_i,PE_j)\in J} t(l_i) & \textit{if pipeline} \\ 1/t_{CNN} & \textit{otherwise} \end{cases} \tag{4.11}$$

$$En_{CNN} = \sum_{(s_i,PE_j)\in J} t * P_{act} + t_{idle} * P_{idle} + b_{acc} * En_{bit} \tag{4.12}$$

In Equation 4.10, the total CNN latency is computed as the difference between end time $s_{|L|}+t(l_{|L|})$ of the last CNN layer $l_{|L|}$ and the start time s_1 of the first one l_1; where layer latency is estimated based on the ALOHA per-layer evaluation procedure, described in Section 4.4.

Equation 4.11 describes the evaluation of throughput for pipeline and sequential schedule. If a CNN is executed as a pipeline, it is evaluated based on the maximum time required to execute the layers mapped on one processor, while for sequential execution the equation considers the network's execution latency.

The total CNN energy En_{CNN} is computed in Equation 4.12, as the sum of the energy costs of all layers. The energy cost of layer $l_i \in L$ accounts for three factors: the inference contribution, obtained as the product of the layer latency and the peak power consumption; the idle contribution, defined according to a latency constraint and the idle power consumption; the cost of bit accesses to the global memory, depending on the number of bits b_{acc} transferred by the processor PE_j during the execution of layer l_i.

4.6 Experimental Results

In the following, we present experimental results involving execution time and energy consumption predictions obtained with the ALOHA method, representing the contribution described in this chapter. In section 4.6.1, we assess the accuracy of our proposed method, in comparison with the OPS count and the Roofline model, estimating the execution time of single layers on both the target platforms, NEURAghe and Jetson, with a reduced average error. In section 4.6.2, we consider an energy consumption model characterized for NEURAghe, and show the impact of accurate execution time and memory access count predictions on the precision of the energy consumption estimation, comparing the ALOHA method and the Roofline model. In section 4.6.3, we assess the impact on a NAS process, aiming at selecting optimal CNN architectures for both target platforms, NEURAghe and Jetson. The last section 4.6.4 considers throughput estimations for CNNs executed on a heterogeneous platform, such as Jetson TX2, with different scheduling schemes. We thus evaluate the combined impact of layer-level ALOHA prediction accuracy and the proposed CNN metric aggregation. All of the considered estimation methods, as well as the aggregation module, and the evolutionary algorithm, were implemented in python3 scripts, running on Azure NC6_v2 Virtual Machine, and exploiting an NVIDIA Tesla P100 GPU.

4.6.1 Layer-level accuracy

To evaluate the ALOHA method in execution time estimation, we defined a grid of over 2000 common CNN convolutional layer configurations, summarized in Table 4.4. We consider as a reference the measured latency value and we quantitatively compare with estimations based

on the OPS count and the traditional Roofline model.

	Parameters
Input Features	*3, 8, 16, 32, 48, 64, 96, 128, 192, 256, 384, 512, 1024*
Output Features	*16, 32, 48, 64, 128, 192, 256, 384, 512, 1024*
Image Size	*2x2, 4x4, 8x8, 14x14, 16x16, 28x28, 32x32,* *56x56, 64x64, 112x112, 128x128, 224x224, 256x356, 512x512*
Kernel Size	*1x1, 3x3, 5x5, 7x7, 11x11*

Table 4.4: Parameters of the convolutional layers measured for the ALOHA method accuracy assessment. The evaluated layer configurations were obtained as different combinations of the listed values, for Input Features, Output Features, Image Size, and Kernel Size.

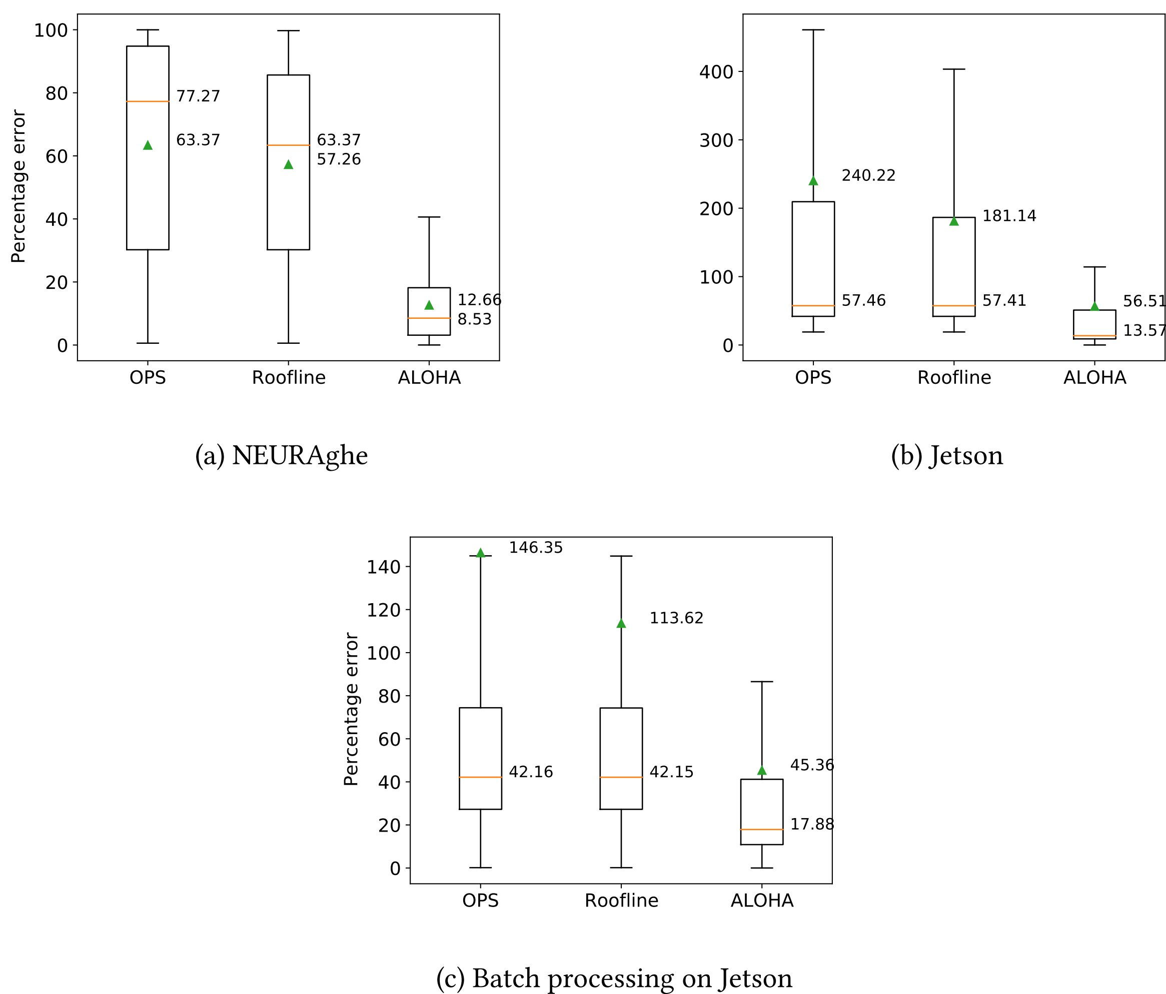
Figure 4.3a and 4.3b show the prediction error distribution for the three methods, through comparison with execution time measured on the target platforms.

NEURAghe.

The execution time predictions are deeply affected by the rounding effects introduced by the computing matrix size, introducing an average 0.25 underestimation (up to a factor of over 0.85) of the actual number of OPS performed during the layer execution. To highlight the contribution of the other non-idealities captured by our approach, the rounding effect correction was included also in the Roofline- and the OPS-based estimations. Nonetheless, as shown in Figure 4.3a, the OPS count method, despite being very immediate and comfortable to build, provides latency estimations having a 63.4% average error. The Roofline model, although introducing rough data transfer time evaluation considering the IO bandwidth ceiling, still shows a 57.3% estimation error. On the other hand, our proposed NEURAghe's ALOHA model proves to be significantly more accurate, reducing the average estimation error to 12.7%.

Jetson.

Compared the hardware-based scheduling in NEURAghe, the runtime management in the GPU engine is intrinsically less predictable. Inefficiencies connected to the Operating System are accounted for in the ALOHA method only in terms of startup time, modeled for Jetson as a



(a) NEURAghe

(b) Jetson

(c) Batch processing on Jetson

Figure 4.3: Error distribution on the latency estimation for the examined estimation methods.

constant startup overhead, thus resulting in a less accurate platform model. The measurements for the accuracy assessment were performed on the TensorRT [107] implementation of the layers in Table 4.4, which is the best-known and state-of-the-art Deep learning library for the NVIDIA Jetson TX2 platform. The estimation error is represented in the boxplot in Figure 4.3b: both the OPS count and the Roofline model provide very poor precision, with up to 2x times over-estimation of the CNN layers performance measured on the platform, while our ALOHA model shows a reduced 56.5% average error.

Impact of batch processing on Jetson.

We finally consider the case of batch processing on the Jetson platform, thus accounting for other processing scenarios. Figure 4.3c reports the estimation error of the examined methods when execution is performed with variable batch sizes, up to a value of 32. This scheduling choice allows for better resource utilization, providing greater opportunities for parallelization. As a result, the measured operating performance is closer to the peak value for the platform, thus both the OPS- and the Roofline-based estimations show a reduced prediction error, although its average value is still over 2$\times$ the one obtainable thanks to the ALOHA method, presented as the contribution of this chapter.

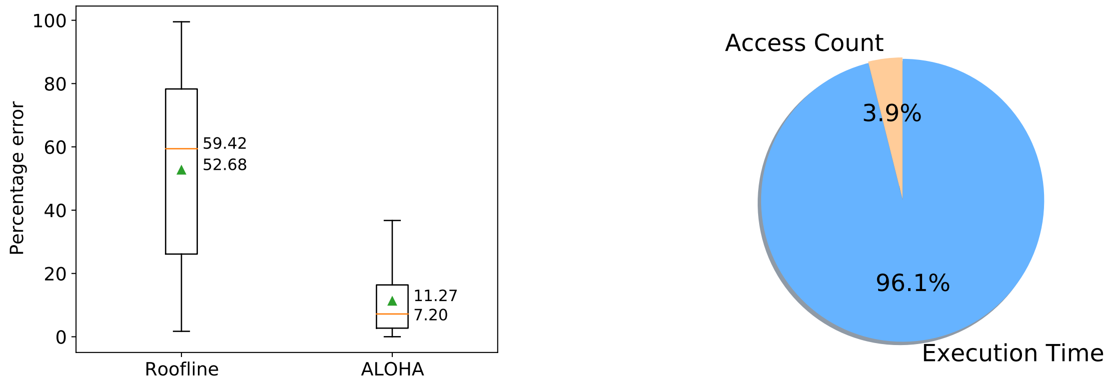
4.6.2 Impact on energy consumption estimation

To assess the impact of our proposed ALOHA accurate modeling on the energy consumption estimations, we consider in this section the dependence of this metric on the execution time and the memory access count, as represented in Equation 4.13, which is a simplified version of 4.12 considering only the power dissipated by a convolutional layer executed on the accelerator:

$$En = P_{act} * t + En_{bit} * b_{acc} \tag{4.13}$$

As we are modeling the execution of a single layer, we neglect here the idle contribution, setting t_{idle}=0 in Equation 4.12. We have thus characterized Equation 4.13 for NEURAGHE, using the values reported in Table 4.2 for P_{act} and En_{bit}. A similar model could be exploited to perform energy consumption estimations for the Jetson, as long as the active and idle consumption values, as well as the energy per bit accessed, are known.

We exploited the simple model in Equation 4.13 to produce an energy consumption estimation for the grid of layers obtained in Table 4.4, considering access count and execution time predictions based on the Roofline and ALOHA methods. Figure 4.4a shows the error distribution, evaluated through the comparison with estimates based on the same model and relying on the measured execution time and the precise access count. The more accurate latency and data transfer evaluations enabled by the proposed ALOHA method produce an average 11.3% estimation error on the energy consumption value. Replacing these metrics with those pro-



(a) Prediction error distribution for the Roofline and ALOHA models.

(b) Contributions to the error in energy estimation based on the Roofline model.

Figure 4.4: Error on NEURAghe energy consumption estimation.

duced by the Roofline model results in an increased average error of around 52.7%. As shown in Figure 4.4b, the error is mostly affected by the predicted execution time, contributing to over 96%. Considering the measured execution time, only a 2% absolute energy estimation error results from neglecting repeated transfers, due to an average 17% error in the memory access count estimations.

4.6.3 Impact on NAS

We analyze here the impact of different degrees of platform awareness on a NAS process. We consider the design of optimal CNN architectures for image classification, targeting CIFAR-10 [45], to be deployed on the target platforms modeled in the previous sections, NEURAghe and Jetson.

The search space is defined based on the structure of the well-known VGG architecture [2]. We restricted the exploration to the CNN topology, neglecting the impact of pre-processing choices and quantization. The total number of considered design points sum up to 3.16M and the considered network architectures are composed as indicated in Table 4.5. Each network presents 5 convolutional stages. Within each stage, all convolutional layers share the same kernel and feature sizes, defined in columns 4 and 5. The architectures differentiate on the

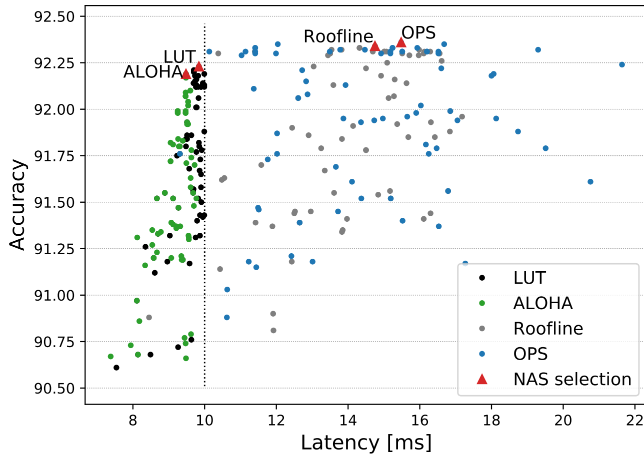
number of convolutional layers in each stage, whose maximum value is reported in column 6, and on their channel width value. Possible width values in each stage are listed in column 3. A MaxPooling layer is placed between successive convolutional stages, while a Global Average Pooling precedes the final GEMM stage, described in the last row of Table 4.5.

Stage	***Operator***	***Output Features***	***Kernel Size***	***Input Size***	***Max Depth***
0	*Conv*	*48/64*	*3x3*	*32x32*	*2*
1	*Conv*	*96/128*	*3x3*	*16x16*	*2*
2	*Conv*	*192/256*	*3x3*	*8x8*	*4*
3	*Conv*	*384/512*	*3x3*	*4x4*	*4*
4	*Conv*	*384/512*	*3x3*	*2x2*	*4*
5	*GEMM*	*384/512*			*2*

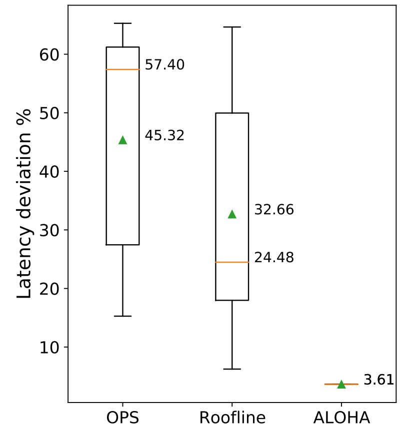
Table 4.5: Design Space Exploration parameters for NAS targeting NEURAGHE and Jetson.

The design space exploration is approached based on the fast selection procedure described in Section 3.2, limited to the first three steps: one-shot training with OFA, population initialization, and exploration based on the GA. The application of a latency constraint defines a restricted platform-aware search space [48], where the network architectures are only admitted if they are compliant with the constraint, based on their latency estimation. The NAS process results in the selection of the most accurate network in the last generation.

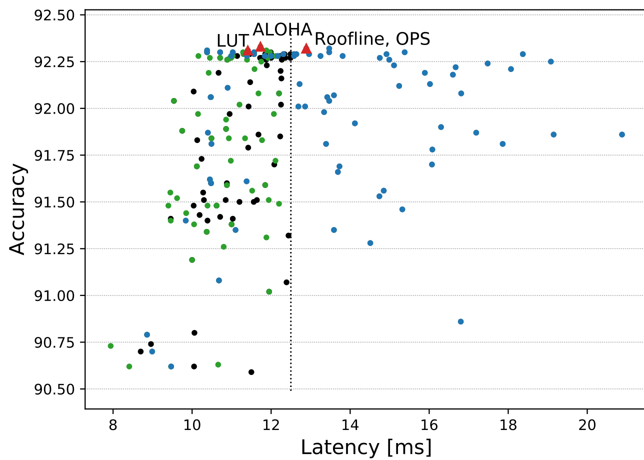
In the following, we define a set of constraints and consider a NAS process targeting the reference platforms. We compare the resulting network selection when the latency evaluation of the design points is performed based on the OPS count, the Roofline model, or our proposed ALOHA method. As a reference for the quality of the final selection, we consider the output of the same process accessing LUTs containing the latency measured on the target hardware. Thus, the four NAS strategies considered (the one based on LUTs, and the ones based on the three estimation methods) define different search spaces and produce after 20 generations an independent selection output. Every NAS experiment is repeated 5 times, to account for the effect of random selections within the genetic algorithm.



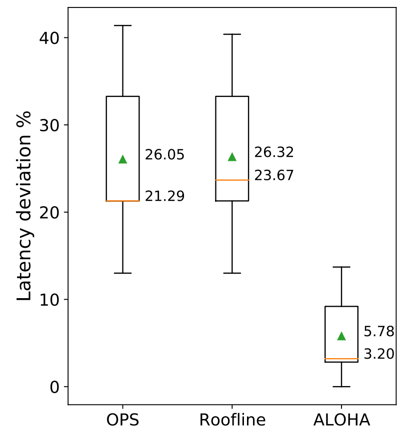
(a) 10 ms



(b) 10 ms

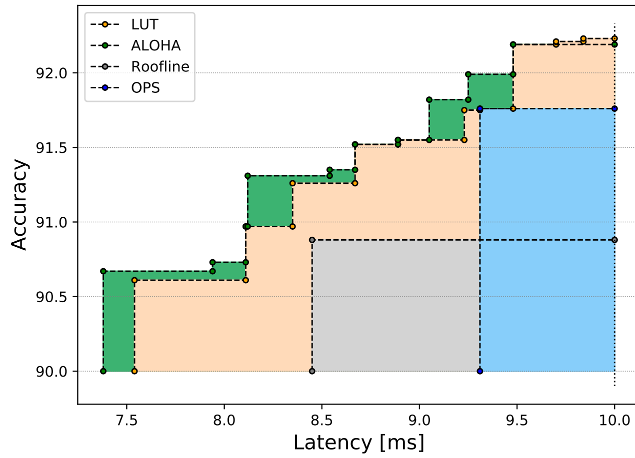


(c) 12,5 ms

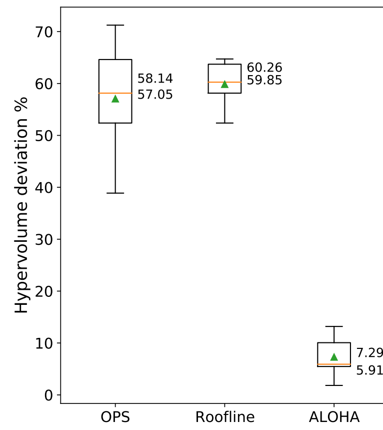


(d) 12,5 ms

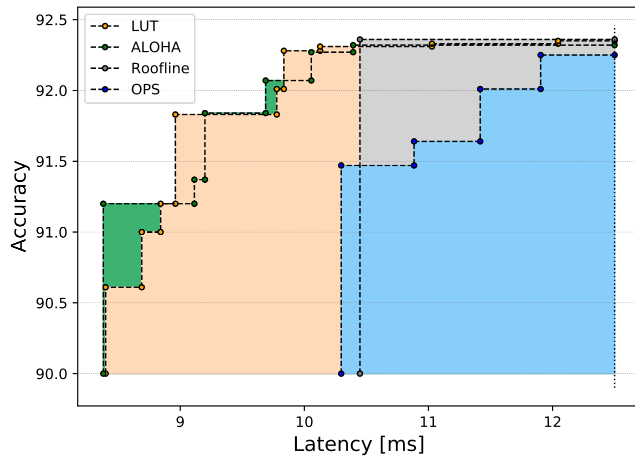
Figure 4.5: a-c)Pareto plot of accuracy vs latency for the design points in the last generation of one trial of evolutionary NAS targeting NEURAGHE. Performance evaluation exploits LUTs or ALOHA, Roofline, and OPS-based estimation. b-d)Latency deviation distribution, among 5 trials, of NAS selection. Performance evaluation based on the ALOHA, Roofline, and OPS models is compared to NAS selection obtained by performance evaluation based on LUTs.



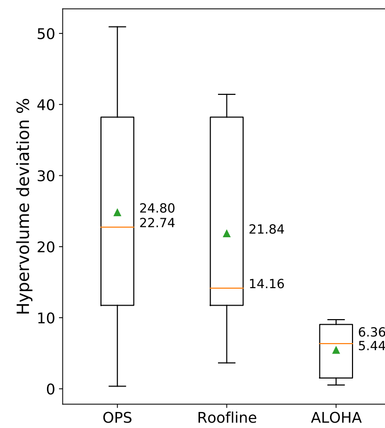
(a) 10 ms



(b) 10 ms



(c) 12,5 ms



(d) 12,5 ms

Figure 4.6: a-c)Hypervolume comparison for the Pareto fronts resulting from evolutionary NAS. Performance evaluation is based on LUTs, and on the evaluated prediction methods. b-d)Hypervolume deviation of the Pareto fronts resulting from NAS based on the examined prediction methods, from the Pareto front produced in NAS exploiting LUTs.

NAS targeting NEURAghe.

We have performed NAS to target two different deployment scenarios, defined by a 10 ms and a 12.5 ms latency constraint. The Pareto plots in Figure 4.5a and 4.5c refer to one of the trials and show the design points belonging to the last generation, placed based on their accuracy and measured latency: the design points selected by the ALOHA-based NAS are much closer to the points selected by the LUT-based NAS accessing direct latency measurements; on the contrary, both the OPS-based and Roofline-based NAS mistakenly focus on complex architectures, reaching higher levels of accuracy, but violating the search constraint on the execution time. However, while the shape of the resulting Pareto fronts in Figure 4.5a for the 10ms constraint looks very different based on the evaluation method considered, the example in Figure 4.5c shows that the inference time prediction accuracy has a lower impact on the design points selection when the latency constraint is more relaxed.

Figure 4.5b and 4.5d, show the difference from the optimal LUT-based selection of the CNN selected by the three estimation-based NAS, in terms of the considered performance metrics (validation accuracy and measured inference latency), summarizing the result of five trials over each constrained space. In general, the ALOHA-based solution is significantly more similar in terms of latency to the optimal LUT-based one (around 3% deviation on average for the 10 ms constraints and always a few percentage points for 12 ms).
To quantitatively estimate the similarity of the explorations, besides the final selection points, we referred to common metrics as the Degree of Approximation [110] and the Hypervolume [111] to compare the Pareto front resulting from each of the NAS processes. As shown in Table 4.6, reporting the DoA values, the ALOHA-driven Pareto front is by one order of magnitude closer to the LUT-driven one, compared to those obtained using the other methods.

Figure 4.6a and 4.6c refer respectively to one NAS trial on the 10ms and 12.5ms latency constraint, showing the hypervolume shapes of the resulting Pareto fronts in the admissible region. The hypervolumes in this section are evaluated by choosing a reference point aligned with the constraint, with coordinates (90%, $constraint$ ms). As can be observed, the pattern resulting from ALOHA-based NAS is similar to the one based on LUTs, while the Roofline and OPS count methods produce significantly different hypervolume shapes. Figure 4.6b and 4.6d

Constraint	*OPS DoA*	*Roofline DoA*	*ALOHA DoA this thesis*
10 ms	*0.54*	*0.63*	***0.02***
12.5 ms	*0.51*	*0.47*	***0.04***

Table 4.6: Degree of Approximation from the reference Pareto front of the fronts resulting from evolutionary NAS based on the examined estimation methods, targeting NEURAGHE with 10ms and 12.5ms latency constraint.

Constraint	*OPS*	*Roofline*	*ALOHA this thesis*
10 ms	*2.5%*	*3.6%*	***100%***
12.5 ms	*26.2%*	*33.7%*	***100%***

Table 4.7: Percentage of admissible design points evaluated in the last generation of evolutionary NAS based on the examined estimation methods, targeting NEURAGHE with 10ms and 12.5ms latency constraint.

report the deviation of the hypervolume indicator from the optimal LUT-based value, throughout the set of trials, confirming that the ALOHA-based Pareto front shapes are overall much more similar to the LUT-based one compared to the alternatives.

Finally, we also compare the methods in terms of predictability and reliability. considering how often the architectures included by the GA in the last population are instead inadmissible according to their on-hardware measured inference time. Table 4.7 reports 100% of admissible points when the exploration is based on our proposed ALOHA method. On the contrary, OPS- and Roofline-based selections include a quite high rate of inadmissible points: only around 3% of the points are legal when the constrain is 10ms and only around 30% when it is 12.5 ms.

NAS targeting Jetson.

In the case of NAS targeting Jetson, we selected a soft latency constraint equal to 3.18 ms, and a more demanding one equal to 2 ms. Although in general the ALOHA method is less accurate on Jetson, it is still able to reduce the discrepancy with the selection operated using LUTs compared to the alternatives, as highlighted in Figure 4.7a and 4.7c. This was observed in all

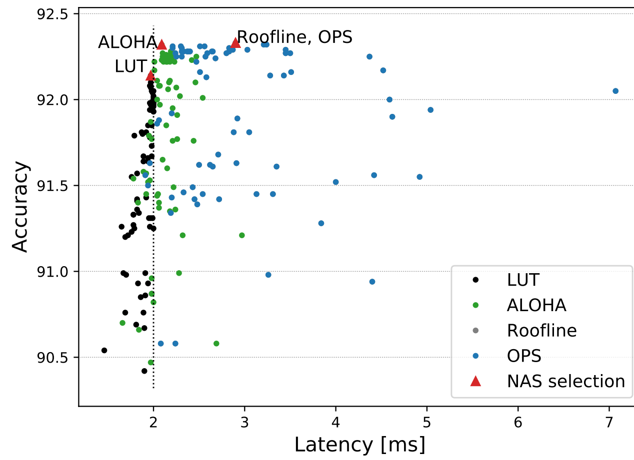
Constraint	*OPS DoA*	*Roofline DoA*	*ALOHA DoA this thesis*
2 ms	*0.3*	*0.32*	***0.07***
3.18 ms	*0.16*	*0.15*	***0.07***

Table 4.8: Degree of Approximation from the reference Pareto front of the fronts resulting from evolutionary NAS based on the examined estimation methods, targeting Jetson with 2ms and 3.18ms latency constraint.

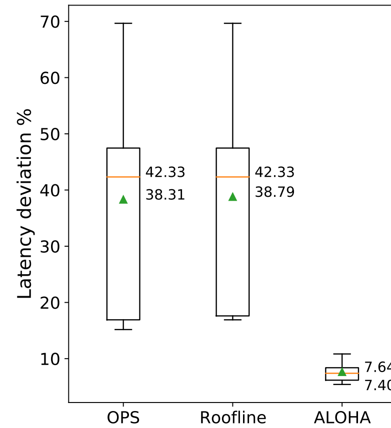
the trials targeting the tightest constraint, as reported in Figure 4.7b, whereas the inaccuracy in performance estimation has a lower impact when the constraint is softened, as shown in Figure 4.7d. Nevertheless, on average, the ALOHA method has enabled a NAS outcome more aligned with the one resulting from the use of LUTs. The same trend is confirmed by the comparison of the Pareto fronts. The hypervolumes in Figure 4.8a and 4.8c show that ALOHA finds Pareto-optimal points that better follow the LUT's Pareto front profile, especially in the case of the 2ms constraint, where the average hypervolume deviation from the optimal one is reduced from over 70% to 30%. The analysis of the DoA metric reported in Table 4.8 provides similar results since ALOHA reduces the deviation by at least a factor of 4. When the constraint is more relaxed the benefits are, as expectable, less visible, although ALOHA still reduces the DoA by 2× and the hypervolume deviation by less than 3×, on average. Table 4.9 provides a view of the effects of our proposed ALOHA method on predictability and reliability, when the exploration targets Jetson. Considering the tightest constraint, when using Roofline and OPS, only around 3% of the architectures in the last population are effectively legal. In this case, also ALOHA is investigating some inadmissible points, however, the rate of legal points at the end of the process is one order of magnitude higher than the other methods. When the constraint is softer, finding admissible points is easier for all the estimation methods, however, ALOHA still proves to be slightly more reliable (76% vs 62%).

4.6.4 Impact of aggregation on prediction accuracy

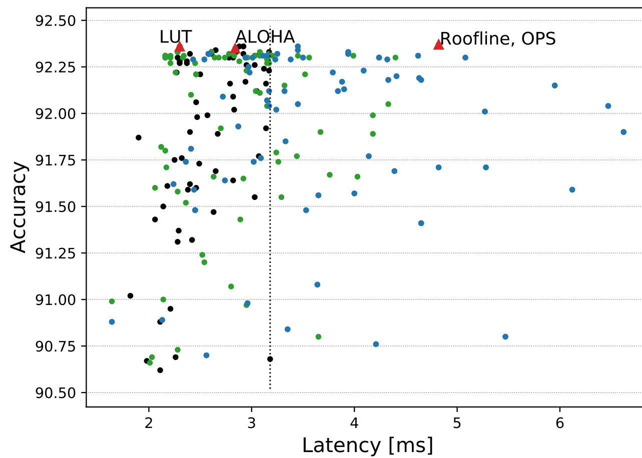
Finally, we evaluate in this section the combined use of layer-level estimations based on the ALOHA method and of the aggregation module, described in Section 4.5 and presented as the



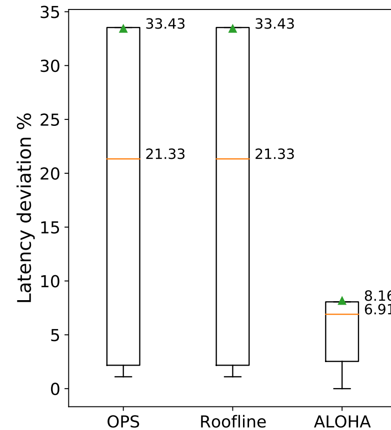
(a) 2 ms



(b) 2 ms

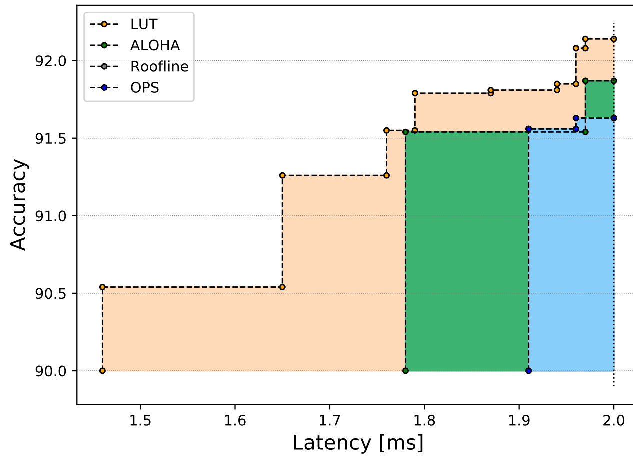


(c) 3,18 ms

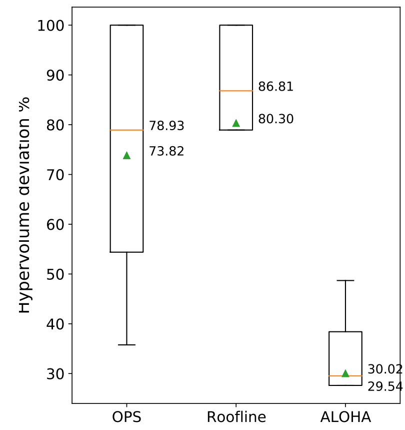


(d) 3,18 ms

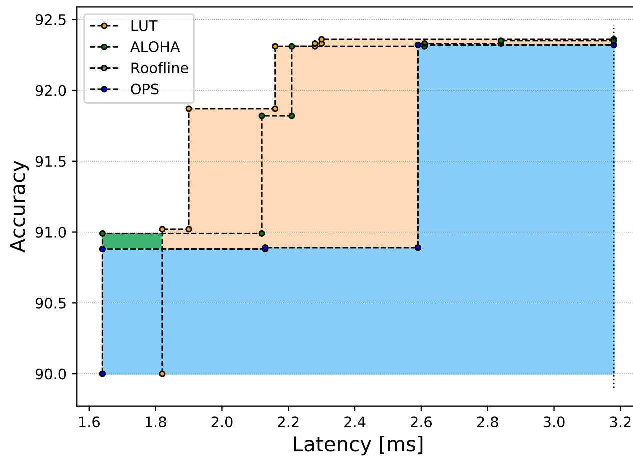
Figure 4.7: a-c)Pareto plot of accuracy vs latency for the design points in the last generation of one trial of evolutionary NAS targeting Jetson. Performance evaluation exploits LUTs or ALOHA, Roofline, and OPS-based estimation. b-d)Latency deviation distribution, among 5 trials, of NAS selection. Performance evaluation based on the ALOHA, Roofline, and OPS models is compared to NAS selection obtained by performance evaluation based on LUTs.



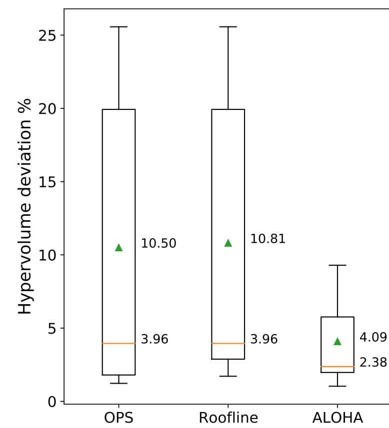
(a) 2 ms



(b) 2 ms



(c) 3,18 ms



(d) 3,18 ms

Figure 4.8: a-c)Hypervolume comparison for Pareto fronts resulting from evolutionary NAS. Performance evaluation is based on LUTs, and on the evaluated prediction methods. b-d)Hypervolume deviation of the Pareto fronts resulting from prediction methods, from the Pareto front produced in NAS exploiting LUTs.

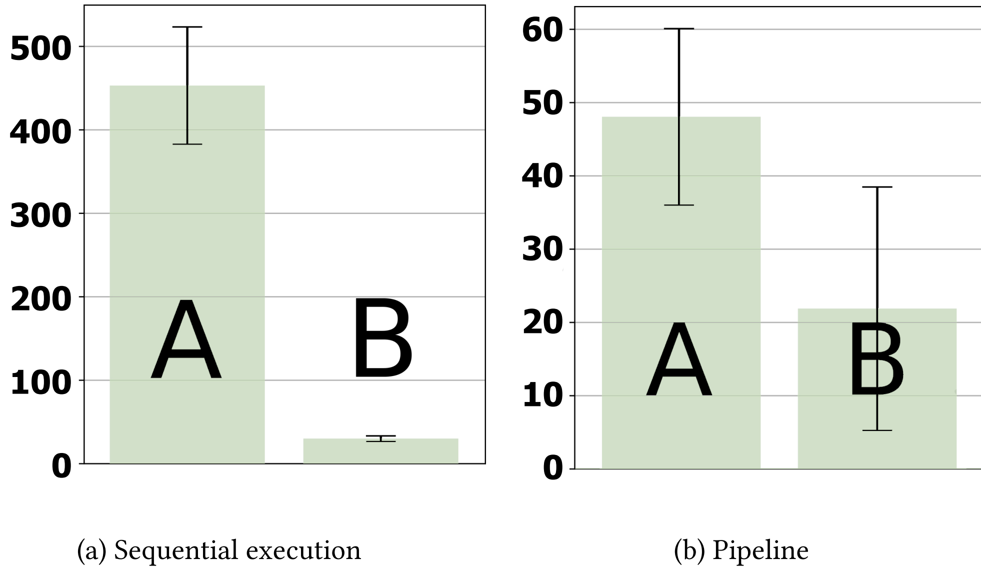
Constraint	***OPS***	***Roofline***	***ALOHA this thesis***
2 ms	*2.6%*	*2.1%*	***29.6%***
3.18 ms	*61.9%*	*61.9%*	***76%***

Table 4.9: Percentage of admissible design points evaluated in the last generation of evolutionary NAS based on the examined estimation methods, targeting Jetson with 2ms and 3.18ms latency constraint.

contribution of this chapter. We consider different CNN execution configurations to estimate the throughput of over 1700 common CNNs, resulting from the NAS exploration described in Section 4.6.3, whose structure is summarized in Table 3.2. We focus on estimating the throughput obtainable on Jetson TX2 heterogeneous embedded platform [26], with different ways of CNN execution. The experiment consists of two trials.

In Trial 1, we study the impact of CNN distribution over platform processors, the GPU and 4 ARM Cortex A-57 CPUs of the Jetson TX2 platform, estimating the throughput resulting from sequential layer execution. We assume a schedule configuration requiring every convolutional layer (where $l_i : op_i = conv$) to be offloaded on the platform GPU, whereas every other layer (having $l_i : op_i \neq conv$) is performed on the platform CPUs. We compare the CNN throughput measured on the platform, with the throughput estimated by the ALOHA method when the CNN execution configuration: A) is unspecified; B) is correctly specified as $pipeline = false$ and $ops_dist = \{(conv, accelerator), (gemm : CPU), (pool : CPU)\}$. The results of this experiment are given in Figure 4.9a representing the mean-and-error. When the layers distribution is not considered, the throughput is estimated with an average error of 450% (estimation A). Taking into account the heterogeneity of the platform and the workload distribution (estimation B) results in a reduced error, as low as 27%.

In Trial 2, we study the impact of considering pipeline execution schedule when evaluating the throughput predictions. We assume layers to be distributed over all the available processors in the platform and we compare the throughput estimation error, with respect to hardware measurements. We consider a CNN execution configuration A) not modeling pipelined execution (setting $pipeline = false$ and $ops_dist = \emptyset$) and B) considering pipelined execution



(a) Sequential execution

(b) Pipeline

Figure 4.9: Error distribution on predicted throughput for CNNs distributed over heterogeneous processors of Jetson TX2

(setting $pipeline = true$ and $ops_dist = \emptyset$). Figure 4.9b summarizes the throughput estimation error at network level. Neglecting parallel execution results in an average estimation error around 48%, which is reduced to 21% when including pipeline-awareness.

4.6.5 Closing remarks

To the best of our knowledge, the estimation method presented in this chapter advances the state-of-the-art of hardware performance modeling solutions, improving the accuracy of common flexible analytical methods, with a reduced modeling effort compared to measurement-based and ML methods. Moreover, the proposed modular structure allows us to take into account the mapping and concurrent execution of different computational kernels on the different processing units available on the platform. Thus, the proposed contribution can provide an easy-to-use solution for the integration of platform awareness in hardware-oriented design. The experiments conducted on the target platforms, NEURAGHE and Jetson, show a reduction of the average latency estimation error by a factor of respectively almost 5× and 3×, when compared to alternatives of similar complexity, such as the Roofline model. We additionally investigated the impact of the improved accuracy in the hardware performance estimation on

the quality of a HW-NAS process. We show that, when performance estimation is based on our proposed ALOHA model, a final network selection well-aligned with the one resulting from a search process accessing LUTs collecting on-hardware latency measurements can be obtained, for different sets of constraints defined on the two target platforms. The average deviation from the hypervolume of the optimal Pareto front is reduced by a factor of $2\times$ up to a factor of $7\times$. Finally, the proposed method can also be exploited to model non-sequential execution schedules.

Chapter 5

Emerging Neural Networks: Optimal Transformer Design

Finally, as emerging network models are joining the landscape of neural networks and edge-processing, there is a need to evaluate and apply efficient hardware-aware design also to these new architectures. The innovative structure of the transformer architecture introduces new parameters to be considered in a design exploration for optimal performance on the target. In the following, we present an example of an efficient transformer model for the monitoring of epilepsy through electroencephalography (EEG) signal processing, resulting from a target-oriented design exploration for deployment on edge devices. As the handling of transformers requires specific support from the optimization tools, the exploration we present in the following does not exploit the benefits of the selection strategy presented in the previous chapters, but it shows that similar concepts still apply to other families of neural networks, as long as the support for the new operands is integrated. To better motivate this statement, we list some common relevant aspects for optimal design:

- *data pre-processing*: the epilepsy monitoring application typically relies on hand-crafted feature extractors; the presented transformer proposes to eliminate this data-dependent extraction step, but the model design is still impacted by data-preparation choices, such as the size of the signal window to be observed with a single inference run;
- *topology*: as the attention layer typically represents the main part of the computational

workload, new parameters should be involved in the exploration, like the number of parallel executions of the attention mechanisms;

- *impact of quantization*, which was not considered as a part of the exploration in this work, but only applied to the finally selected design.

The study presented in the following was conducted thanks to the collaboration with Zurich's ETH (responsible for the funding PEDESITE project, supported by the Swiss National Science Foundation) and the University of Bologna. To summarize the outline of the chapter, in Section 5.1 we present the epilepsy monitoring problem and its challenges, motivating the need for a new efficient small-scale seizure detector design. In Section 5.2 we describe our proposed transformer model, the EEGformer, and two CNN-based detectors we consider as a reference for performance comparison. In Section 5.3 we assess its performance on the CHB-MIT Scalp EEG dataset [112, 113] and define a comparison with the state of the art of seizure detectors. Finally, we present in Section 5.4 the inference performance on a tiny MCU, the Ambiq Apollo4, and on the multi-core PULP processors GAP8 and GAP9, by Greenwaves. The measurements conducted on the three platforms considered show the target-oriented design resulted in an efficient model, which can be run on a low-power monitoring device with as low as 13.7ms inference time and 0.19mJ energy consumption.

5.1 Problem definition

According to the World Health Organization, more than 50 million individuals worldwide suffer from epilepsy [114]. The use of antiepileptic drugs usually limits its impact on the quality of life, however, this treatment isn't always effective in preventing the occurrence of seizures, causing the patients to lose control over their bodies and representing a danger to everyday life. The analysis of the electroencephalography (EEG) signal represents the usual diagnostic instrument and provides an opportunity for the development of monitoring solutions, based on automatic seizure detection. The design of a suitable detector needs to satisfy both accuracy and hardware constraints, resulting not only from the limited computational and storage resources of a low-power processing unit but also from a size factor compatible with an un-

obtrusive signal acquisition. The tolerability of a similar device, conditioning its practical use, depends on the possibility to disguise it with behind-the-ear accessories [115, 116, 117] and on the reliability of the alarms raised.

The literature reports several examples of very accurate detectors [118, 119, 120], reaching good performance metrics but relying on a complete acquisition setup [121, 122, 123, 92, 124, 125, 126, 127, 128]. Moreover, the most common approaches to EEG processing require a feature extraction step: the work of [115] is an example of wavelet-based pre-processing, providing energy values as input to the classifier. As the clinical practice involves data volumes incomparable with the currently limited datasets available for research (the basis for the design of feature extraction), we propose to exploit the learning process and rely on a classifier working on the raw signal. The attention mechanism in the transformer represents a promising solution.

The use of transformers for epilepsy monitoring has been previously investigated in [124, 125, 92]. However, these works do not target unobtrusive acquisition, as they consider complete electrode setups, not suitable for long monitoring. Moreover, they propose large-scale transformers, especially in [124] and [125], whose complexity and footprint are out of scale for embedded deployment. As for the work of [92], it doesn't explicitly target edge processing and a clear indication of the complexity of the detector is not given. While [124, 125] rely on pre-processing based on Fourier transform, we mean to work on the raw signal, as in [92].

To the best of our knowledge, seizure detection based on unobtrusive low-channel count acquisition is still rather unexplored and the best reference for the assessment of the detection results is the work of [115], performing detection based on the EEG signal acquired by the temporal channels and exploiting wavelet-based energy features. This work shows very promising results on a subset of the CHB-MIT scalp EEG dataset [112, 113], presenting an AdaBoost (AB) model reaching up to 100% accuracy, but with an onset detection latency of 19s, which still requires improvement. To consider the general state-of-the-art context, we will also compare to [128, 127, 126], presenting detectors based respectively on CNNs, K nearest neighbor (KNN), and Support Vector Machines (SVMs), and exploiting complete acquisition steups. These works provide to the best of our knowledge the best tradeoff between high sensitivity (up to 98%) and detection rate, and nearly zero false alarm rate (best-reported value is

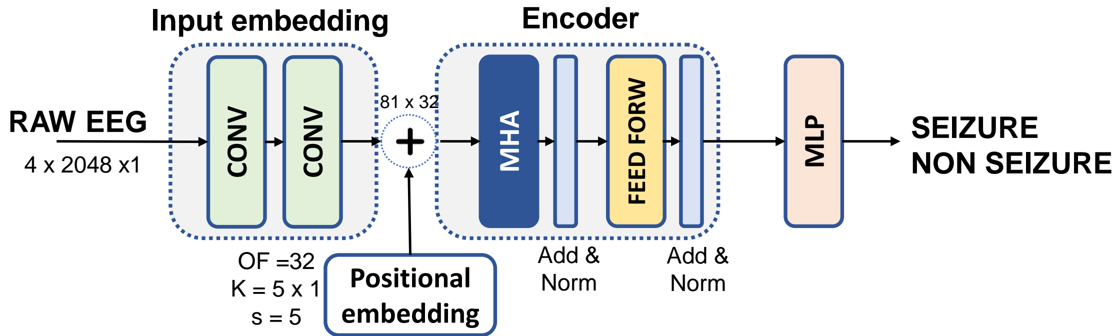


Figure 5.1: Architecture of the EEGformer.

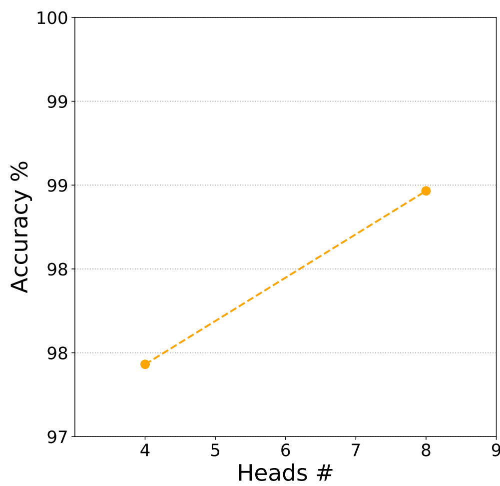
0.2 FP/h).

5.2 Detector

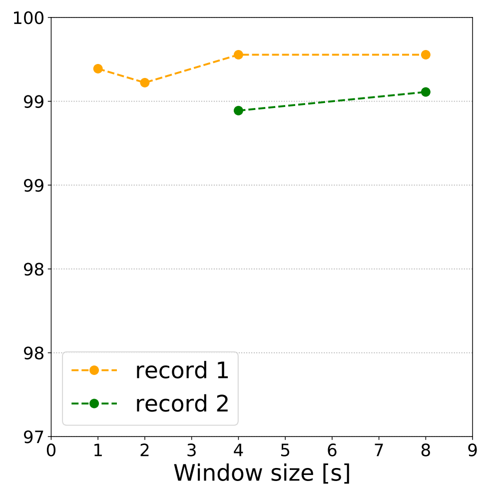
We treated the seizure detection task as a simple classification problem applied to sliding windows of the signal, where the detector needs to evaluate whether the processed sample belongs to the non-seizure or to the seizure class. Considering the limited state-of-the-art references based on low-channel count acquisition, we optimized two CNN structures for the seizure detection task on the CHB-MIT dataset, to provide a more complete assessment of the performance of our proposed model, the EEGformer. The structure of the EEGformer is described in Section 5.2.1, whereas in Section 5.2.2 we describe a CNN-based detector, operating on the raw EEG signal, and in Section 5.2.3 one operating on features representing the energy after wavelet decomposition. As we target unobtrusive acquisition setups, we only consider data acquisition from the temporal channels: F7-T7, T7-P7, F8-T8, T8-P8, according to the 10-20 international system.

5.2.1 EEGformer.

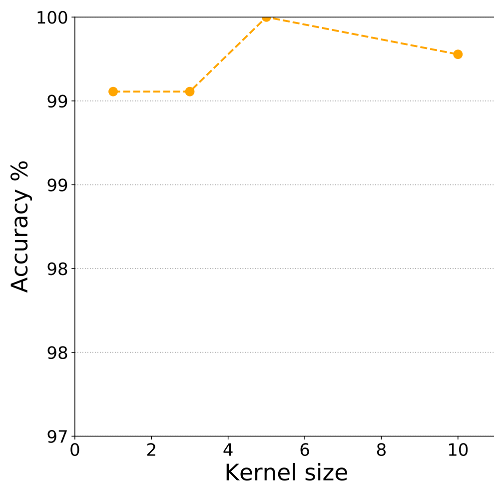
The EEGformer is configured as a Vision Transformer (ViT), which is the most common transformer structure for classification problems. The input image is replaced by a 4-dimensional tensor, organizing the samples of one window of signal acquired by the temporal channels. The most important processing stage is represented by the encoder structure, where the attention



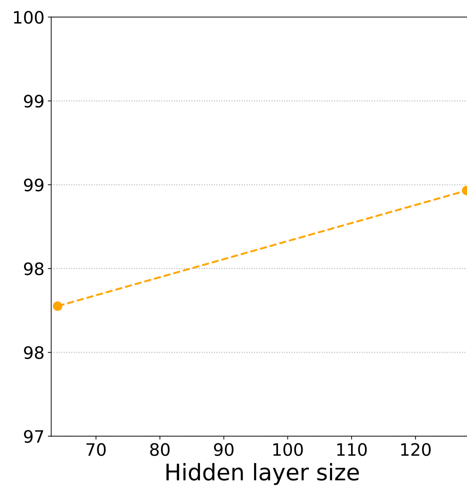
(a) # heads: 4, **8**



(b) window size: 1s, 2s, 4s, **8s**



(c) kernel size: 1, 3, **5**, 10



(d) hidden size: 64, **128**

Figure 5.2: ViT architecture parameter exploration. The selected solution is highlighted in bold.

mechanism is applied. The attention principle [86] is defined in Equation 5.1 and allows to evaluate the dependencies between two different points in the time series: *query* q, *key* k and *value* v represent different linear projections of the input sequence, and d is the size of each projection.

$$Attention(q,k,v) = Softmax(\frac{qk^T}{\sqrt{d}}) * v \tag{5.1}$$

To be properly processed by the attention layer, the input data needs to be arranged as a sequence of patches. Exploiting image classification as a clear example, each patch represents a tile of the image, while the dimensionality of the sequence can be adjusted based on the parameters of the first embedding stage, which is represented by a pair of convolutional layers in the EEGformer. The simple idea of the attention can be upgraded considering different projections of the input sequence evaluated in parallel. As every parallel execution is known as a $head$, this implementation is known as Multi-Head-Attention (MHA). The encoder block is terminated with a feed-forward network, while the classification stage is obtained with a Multi-Layer Perceptron (MLP).

As will be discussed in Section 5.4, the MHA represent the most computational intensive portion of the transformer. We thus focused on defining the optimal parameters impacting its size, to obtain an architecture suitable for efficient execution on tiny MCUs. Considering as a reference a maximum footprint of 512KB, based on the commonly available storage space, we performed a design exploration of the architectural parameters, aiming to optimize the accuracy on the CHB-MIT dataset, where the EEG signal is acquired with a 256Hz sampling frequency. Figure 5.2 reports the details of the exploration. We evaluated the size of the input window, selecting a length of 8s, resulting in a 4$\times$2048$\times$1 input size (Figure 5.2b). The shape of the attention input was explored as the result of the embedding stage: as shown in Figure 5.2c the best results were obtained with a kernel size of 5 and the same value for the stride, in convolutional layers with 32 output channels. To refer to the common transformer terminology, the attention is thus performed on a sequence of embeddings of size 81$\times$32, while the value selected for the projection size is $d = 32$. The exploration also involved the number of heads in the MHA, resulting in the selection of $H = 8$, as shown in Figure 5.2a. Finally the MLP for classification has a hidden layer size of $h = 128$ 5.2d. The final structure of the EEGformer is shown in Figure 5.1: it requires 50.6K parameters and 14.7 MOPs.

5.2.2 CNN on raw EEG signal.

As a first model for comparison we optimized a CNN architecture operating on the raw EEG signal. The first stage is very similar to the EEGformer's, exploiting windows of 8s length and a convolutional layer shaped in the same way as the EEGformer's. The structure of the network is summarized in Table 5.1: the model consists of two convolutional layers (Conv#), followed by Rectified Linear Units (ReLU) activation functions, a MaxPooling and two Fully Connected (FC#) layers. It has a memory footprint of 325KB, evaluated considering 8-bit representation, and involves 2.22MOPs. We refer to this model as CNN B.

5.2.3 CNN on pre-processed input features.

Finally, we considered a reference model to evaluate the impact of signal pre-processing. The CNN C is summarized in Table 5.2: its input is obtained with 8-level wavelet decomposition, and represents the energy of the signal evaluated on each temporal channel during 8 different time frames. Every time frame is a window of 8s length, and is partially overlapped to the others with a 1s step size. The input shape is thus 4×8×8, corresponding to $channels \times levels \times frames$. The architecture exploits a sequence of 5 convolutional layers, followed by ReLu activation. A MaxPooling layer reduces the dimensionality prior to the last FC module. The network requires 12.5 MOPs and 105.3KB of parameters.

Layer	***Input Features***	***Output Features***	***Input Size***	***Kernel Size***	***Stride***
Conv1	*4*	*32*	*1x2048*	*1x5*	*5*
Maxpool	*32*	*32*	*1x405*	*1x2*	*2*
Conv2	*32*	*32*	*1x203*	*1x5*	*2*
Maxpool	*32*	*32*	*1x102*	*1x2*	*2*
FC1	*1632*	*200*	*1x1*		
FC2	*200*	*2*	*1x1*		

Table 5.1: CNN architecture with raw signal input

Layer	*Input Features*	*Output Features*	*Input Size*	*Kernel Size*	*Stride*
Conv1	*4*	*16*	*8x8*	*3x3*	*1*
Conv2	*16*	*32*	*8x8*	*3x3*	*1*
Conv3	*32*	*64*	*8x8*	*3x3*	*1*
Conv4	*64*	*64*	*8x8*	*3x3*	*1*
Conv5	*64*	*64*	*8x8*	*3x3*	*1*
Maxpool	*64*	*64*	*8x8*	*2x2*	*2*
FC1	*1024*	*2*	*1x1*		

Table 5.2: CNN architecture with pre-processed input

5.3 Assessment on CHB-MIT dataset

We assessed the quality of seizure detection with the EEGformer considering the CHB-MIT scalp EEG dataset [113, 112], which provides a meaningful common benchmark for a comparison with the state of the art. We first summarize the evaluation of two different training strategies, to select the most effective for the assessment. Finally, we compare the performance of the EEGformer with the CNN alternatives described in Sections 5.2.2 and 5.2.3 and with the state of the art, considering low-channel count unobtrusive acquisition and complete acquisition.

	Evaluation Period	*Sensitivity; Specificity*	*FP/h*	*Detected Seizures*
without Pre-Training	*2s*	*95.4; 99.9*	*0.9*	*7/7*
Pre-Training	*2s*	*86.6; 100*	*0*	*7/7*

Table 5.3: EEGformer training strategy evaluation

Training strategy. First, we assess two different training approaches: a one-phase method consisting of subject-specific training, lasting 100 epochs, and a two-phase strategy including a global pre-training stage (100 epochs) and a subject-specific fine-tuning (50 epochs). For every subject, the dataset provides a list of EEG records, some of which report a seizure event. We focus on testing such records, with the leave-one-out approach, so that the test set is alternatively represented by one seizure record. The remaining data is randomly split between

a train set (80%) and a validation set (20%). The data is arranged into 8s-wide windows: in the training set the windows are not overlapped, while for testing we consider overlapping windows with 2s step size, to improve the detail of the performance evaluation.

The two training approaches are compared in Table 5.3, referring to tests conducted on subject 1, with pre-training conducted on the data of subjects 2 to 8. As can be observed, the pre-training phase improves the specificity of the detector, resulting in 0 false-positive-per-hour (FP/h) rate, which is a very relevant metric for encouraging the use of a monitoring device. Despite a drop in sensitivity, all the annotated seizure events were detected, we thus selected the two-phase approach as more effective.

Detection performance assessment. At this point, we compare the EEGformer with the optimized CNN alternatives presented in Section 5.2. Considering the benefits resulting from the global pre-training phase, we applied this training approach to all three of the considered detectors. For the performance assessment, we consider a subset consisting of 8 subjects. Thus, the pre-training step is performed based on the data of seven patients, excluding the test subject. For each one, we assess with the leave-one-out approach the performance on each seizure record and consider a cumulative evaluation of the most relevant quality metrics. Based on a typical approach in the literature, we perform post-processing of the classifier output to limit the number of FPs: we consider the minimum number of windows to be averaged in order to improve the specificity, which is 3 for the EEGformer, and 5 for both CNN B and C. Moreover, as the instability induced by the seizures on the signal lasts several minutes, we neglect the FPs occurring within 15 minutes after the end of the episode. Table 5.4 summarizes the results, including the state-of-the-art AB model assessed on the same subset of subjects, with equivalent 3 windows averaging. The comparison highlights that no significant degradation results from the elimination of the feature-extraction step. On the contrary, we report in the second line the metrics of the EEGformer if applied after an artifact removal stage, as the one described in [129]. In this case, the performance is well-aligned with the state-of-the-art of low-channel count detectors. We analyze in the following the numbers obtained with the EEGformer. Figure 5.3 reports the distribution of the FP/h trend across the 40 tests performed: the median value is 0 FP/h, which is the false alarm rate registered in over 80% of the left-out records tested. The non-zero average is the result of 5/40 outliers and is mostly affected

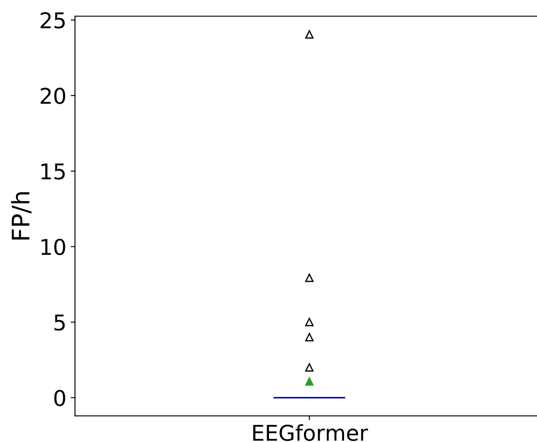


Figure 5.3: Record-wise FP/h signaled by the EEGformer on the CHB-MIT dataset.

by the two uppermost points in the plot. The analysis of the corresponding records showed long-lasting EEG artifacts causing the FP occurrence, whose removal would produce the data reported in the second line. Figure 5.4 shows the detailed detection rate on every patient: 100% of the annotated events are successfully detected on 6/8 subjects. The overall sensitivity is compromised by the poor performance obtained on patient CHB 6, presenting seizures of a very short duration. Overall, the EEGformer reaches quality metrics comparable to the state-of-the-art reference for unobtrusive detection based on the temporal channels, without requiring handcrafted feature-extraction and introducing a 20% reduction in the average onset detection latency. Lower latency is reported in other works, such as [130, 131], but at the expense of a lower specificity, which is crucial for the tolerability of a continuous monitoring device.

For the sake of completeness, we also report in Table 5.5 the general state of the art, referring to the performance metrics reported in the corresponding papers and considering unconstrained models providing the best trade-off between very high sensitivity and near-zero false alarm rate. We omit the size of the models in [127, 126], which is not explicitly reported. A general analysis of the numbers reveals a performance gap compared to the results achievable with access to complete acquisition setups, which encourages new advancements.

	Evaluation Period	*Sensitivity*	*Specificity*	*FP/h;*	*Average Latency*	*Detected Seizures*
EEGformer	*2s*	*65.5*	*99.9*	*0.8*	*15.2s*	*32/44*
EEGformer[*1]	*2s*	*66*	*99.9*	*0.12*	*15.2s*	*32/42*
CNN B	*2s*	*65.3*	*99.9*	*2.8*	*18.2s*	*32/44*
CNN C	*2s*	*53.5*	*99.7*	*8.2*	*22.6s*	*30/44*
AB ([115])	*4s-8s*	*72*	*99.9*	*0.5*	*19s*	*38/44*

Table 5.4: Performance comparison on CHB-MIT dataset considering acquisition from temporal channels.

	Sensitivity	*Specificity*	*FP/h*	*# acquisition channels*	*needs pre-processing*	*# params*
EEGformer	***65.5***	***99.9***	***0.8***	***4***	✗	***50.6 K***
EEGformer[*1]	*66*	*99.9*	*0.12*	*4*	✗	*50.6 K*
AB ([115])	*72*	*99.9*	*0.5*	*4*	✓	*4 K*
SVM ([126])	*97.34*	*97.5*	*0.63*	*18-23*	✓	-
KNN ([127])	*98.4*	*99.1*	-	*18-23*	✓	-
CNN ([128])	*88.14*	*99.62*	*0.2*	*18-23*	✗	*105 K*

Table 5.5: Comparison with the state of the art on the CHB-MIT dataset.

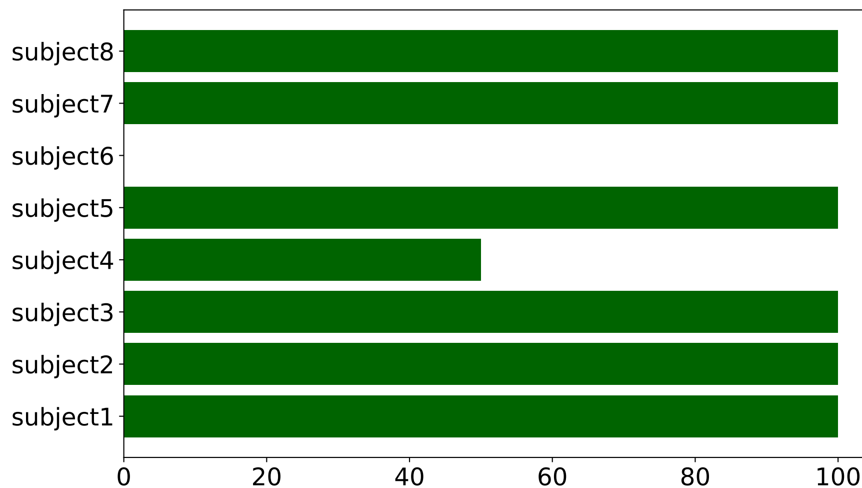


Figure 5.4: Percentage of detected seizures per patient with the EEGformer on the CHB-MIT dataset.

5.4 Deployment

Finally, we assessed the efficiency of detection based on the EEGformer, running on three different resource-constrained platform. The reported results show that the EEGformer is suitable for real-time monitoring on low-power devices. To improve the efficiency of the implementation, we performed quantization to 8-bit, thus reducing the memory footprint and enabling byte-level processing, exploiting the Quantlab software package for quantization-aware fine-tuning [132]. For every target, we considered the most energy-efficient setup of voltage supply and working frequency.

5.4.1 Deployment on Apollo4

We first considered a single-core platform, the Ambiq Ultra-Low-Power Apollo4 MCU [28]. Its computing resources are represented by a 32-bit ARM Cortex-M4 processor accessing a 2MB MRAM and a 1.8MB SRAM. It is particularly suitable for health-monitoring applications, because of its power efficiency (5μA/MHz) and the possibility to tune the clock frequency based on the application's requirements. We exploited the CMSIS-NN library [34] to obtain an efficient implementation of the attention layer, as it is described in [133]. We report the inference metrics in the first column of Table 5.6. Having tuned the frequency to 96MHz, we measured 405 ms inference time and 1.79mJ energy consumption, based on the average power consumption measured with the Keysight N6715C DC power analyzer.

5.4.2 Deployment on GAP: exploit parallelism

The parallel structure of MHA offers the opportunity to improve the inference performance through parallel execution. We thus considered deployment on a multi-core platform, embedding a cluster of parallel RISC-V processors. The targeted devices are the GAP8 [134] and GAP9 [29] platforms, from Greenwaves.

The GAP8 integrates nine RISC-V parallel processors, with one serving as a control processor (Fabric Controller), and eight exploited as the computing cluster. The voltage and working frequency of the cluster can be adjusted in accordance with the application's requirements. The memory hierarchy includes a 512KB L2 shared memory and a small 64KB L1 memory which

is local to the computing cluster. Multiple DMAs enable independent and power-efficient data transfers. This processor is built with the 55 nm TSMC LP technology, which supports clock frequencies up to 250 MHz. As demonstrated in Figure 5.5, the MHA layer represents the primary computational workload, which we distribute on the computing cluster, along the $heads$ dimension. The implementation is inspired to the one described in [133], exploiting a dataflow similar to the one enforced in the CMSIS library, efficiently grouping sets of multiply-and-accumulate operations to be completed in a single clock cycle. Figure 5.6 illustrates as this solution enables an almost linear speedup with the number of cores. We selected the most energy-efficient configuration, with 65MHz clock frequency and 1V voltage supply, and evaluated inference performance. The results are reported in Columns 2 and 3 of Table 5.6. Despite single-core execution not being more energy efficient than the first implementation on Apollo4, parallel execution allows for over $3\times$ speedup in the overall inference time, resulting in a 30% lower energy consumption per inference.

With an operating frequency of up to 400MHz and a design based on 22nm TSMC LP technology, the GAP9 processor represents a more advanced technological node. The available storage space is larger, providing a 128KB L1 memory and a 1.6MB L2 memory, and the computing cluster is enriched with an additional supervising core. As demonstrated in the plots in Figure 5.7b and 5.7c, the most power efficient setup for the platform is obtained with 1.8V supply voltage and 240MHz: the power consumption is reduced by a factor of 1.6 for parallel execution on 8 cores, resulting in $1.11\times$ energy saving. Thanks to the parallel execution on the computing cluster, we are able to obtain an inference time that is equal to 22% of the one required by GAP8, with 89% energy savings over the original Apollo4 implementation. Further improvements to the parallel implementation, delivering a higher performance gain, are still possible.

5.4.3 Closing remarks

The main design knobs available for the optimization of CNNs define a similar design space also for other emerging neural models, such as the transformer. In this chapter, we presented the hardware-oriented optimization of a transformer-based seizure detector, obtained considering the storage constraints of common commercial micro-controllers and reaching perfor-

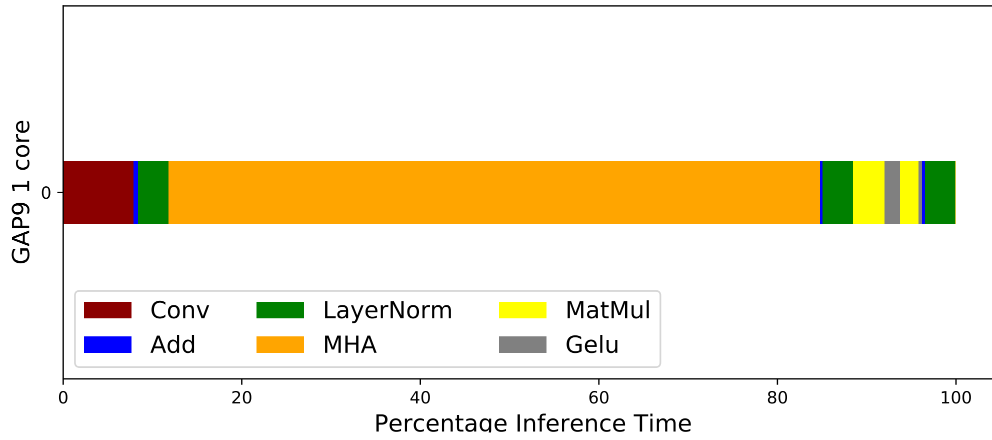


Figure 5.5: Percentage inference time distribution for the different operators in EEGformer on GAP9 and 1 core execution.

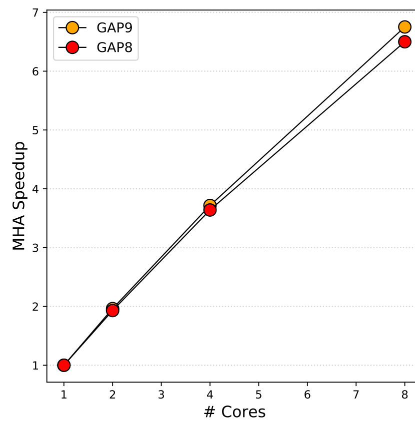
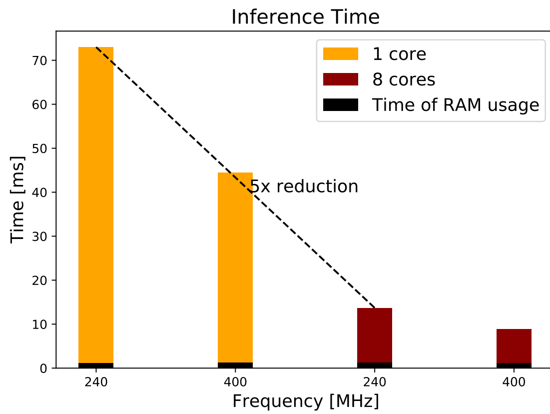
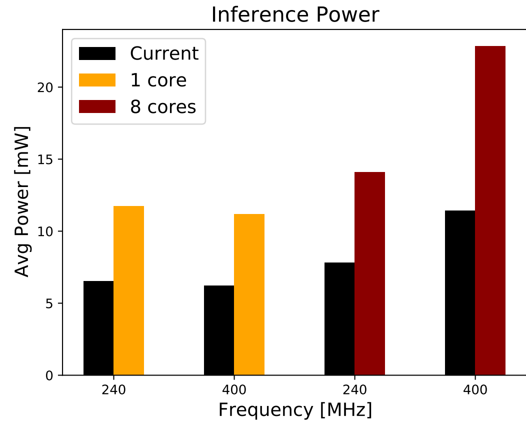


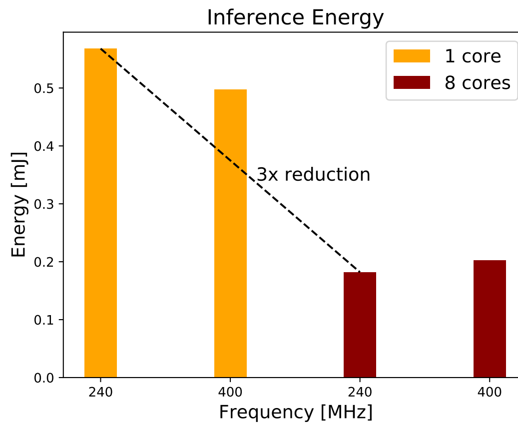
Figure 5.6: Parallel speedup of the MHA execution time for execution on the GAP processors.



(a) Inference time on GAP9 at different clock frequencies for single-core and parallel execution.



(b) Inference power on GAP9 at different clock frequencies for single-core and parallel execution.



(c) Inference energy on GAP9 at different clock frequencies for single-core and parallel execution.

mance metrics well-aligned with the state of the art. The different choices of input data size and topologies compared in this chapter constituted a restricted set of alternatives. Nonetheless, a design approach based on HW-NAS, as the one described in Chapter 3, can still be applied to this family of networks, providing the necessary updates to the tools described. In particular, the most relevant challenge is the integration of the support for the MHA layer on the different training engines, as well as on the quantization and code generation tools. Moreover, since the MHA represents the main computational workload in the transformer architecture, the latency modeling tool should also be updated to correctly estimate the hardware performance of this particular operand, whose impact on the hardware inference efficiency is indeed non-negligible. As MHA is configured as a set of simpler operands, it should be considered whether its modeling would require the introduction of a new line in Table 4.1, representing the MHA operand, or whether it would be more convenient to separately model its components and aggregate the final performance estimation.

	Apollo4	***GAP8***		***GAP9***	
	1 core	***1 core***	***8 cores***	***1 core***	***8 cores***
Frequency	*96 MHz*	*65MHz*	*65MHz*	*240MHz*	*240MHz*
Time/inference	*405ms*	*283.9*ms	*62.2*ms	*72.99ms*	*13.7ms*
Total Power	*4.4mW*	*10mW*	*18.1mW*	*11.75mW*	*14.11mW*
Total Energy/inference	*1.79mJ*	*2.9mJ*	*1.2mJ*	*0.57mJ*	*0.19mJ*

Table 5.6: Inference performance on hardware.

Chapter 6

Conclusions

We have discussed the main challenges of optimal neural network design for embedded edge-processing and presented our research effort to provide guidelines and methodologies to efficiently address them.

As a first result, we described the ALOHA design flow, as an efficient method for the evaluation of large design space explorations aiming at optimal design for CNN-based edge systems. It is able to combine the network topology and data pre-processing exploration, provide target awareness, and consider different quantization levels. We have proposed a fast implementation, to obtain near-optimal results with a reduced exploration time, and a more detailed one, including a more precise characterization phase to refine the performance evaluation considered during the exploration with the impact of quantization on the final CNN model accuracy. We have considered as a reference use case the design of CNN classifiers for a KWS task to be deployed on the SensorTile MCU and tested our selection procedure on two deployment scenarios defined by constraints derived from the state of the art [61, 62]. We showed how the proposed automated procedure allows us to design specifically tailored network models, whose performance is comparable with the state of the art of CNNs for KWS: with around 30h exploration (conducted on an NVIDIA Tesla T4 GPU and an NVIDIA Tesla P100 GPU) we obtained a CNN model improving by 1.8% the accuracy of [61], with 40% less required storage space for weights and activations.

We further focused on the impact of platform awareness on the quality of a target-oriented design based on NAS. Considering the need of integrating an accurate and reliable hardware

performance evaluation in the search problem, we developed the ALOHA estimation method to improve the accuracy of inference modeling, especially when heterogeneous and parallel platforms are targeted. We have tested the accuracy of the obtainable performance estimations considering the modeling of two target platforms, the FPGA-based NEURAghe accelerator, and the GPU-based Jetson TX2 platform. The proposed method is more accurate than the others considered for the comparison, having a similar level of complexity and development effort, namely OPS count and the Roofline model, and can be easily integrated into a NAS process, thus eliminating the need to train complex ML-based predictors or having to include the hardware-in-the-loop. The average error in latency estimation is reduced by 3×, up to 5×, while the precision of the energy consumption estimation is improved by 2×. We evaluated the impact on a NAS process selecting optimal CNN models for image classification, considering both reference platforms and a set of throughput constraints. Comparing the output of independent search processes where the performance evaluation exploits our ALOHA method, OPS count, or the Roofline model, and considering as a reference an optimal scenario where direct measurements of latency stored in LUTs can be accessed, we show that the ALOHA method allows obtaining selection outputs very similar to the optimal case, resulting in a 4× higher predictability of the search process.

Finally, moving forward from the restricted CNN design space, we consider the optimal target-aware design of a transformer-based seizure detector, to be deployed on low-power unobtrusive health-monitoring devices. We addressed the design space exploration considering the typical storage resources of tiny MCUs and explored the most relevant parameters impacting the footprint and complexity, as well as the accuracy, of the proposed transformer model. Our resulting architecture, the EEGformer, reaches performance metrics well-aligned with the state of the art of low-channel count detectors on the CHB-MIT dataset and it is not too far from unconstrained detectors. It does not require any handcrafted feature extractor and it is able to detect 73% of the examined seizure events, with 100% specificity on 35/40 tests. Finally, we exploit quantization up to 8-bit precision and show the inference performance on three different hardware targets, the Apollo4 MCU and two PULP-based processors of the GAP family, GAP8 and GAP9. The EEGformer can be executed with as low as 13.7ms and 0.19mJ and is thus suitable for deployment on tiny monitoring devices.

At this point, while CNN optimization benefits from pretty efficient target-oriented design and optimization approaches, a step forward is needed to extend these benefits to emerging architectures. The presented epilepsy monitoring use case represents a meaningful example of an application requiring the efficient design of a transformer detector. The automation of the design space exploration raises new challenges, due to the new operands and topologies considered. Furthermore, the classification accuracy on a reduced validation set is not always a meaningful metric for pre-evaluating the performance of the candidate design points. When highly unbalanced problems, like the epilepsy monitoring task, are addressed, a more complex definition of the target metric would be needed, where different cross-validation approaches are considered, and the accuracy metric is replaced by more suitable ones for the evaluation of unbalanced cases, such as sensitivity and specificity.

The work described in this thesis produced the following publications:

- P. Busia, S. Minakova, T. Stefanov, L. Raffo and P. Meloni, ”ALOHA: A Unified Platform-Aware Evaluation Method for CNNs Execution on Heterogeneous Systems at the Edge,” in IEEE Access, vol. 9, pp. 133289-133308, 2021, doi: 10.1109/ACCESS.2021.3115243.

- P. Busia, G. Deriu, L. Rinelli, C. Chesta, L. Raffo and P. Meloni, ”Target-Aware Neural Architecture Search and Deployment for Keyword Spotting,” in IEEE Access, vol. 10, pp. 40687-40700, 2022, doi: 10.1109/ACCESS.2022.3166939.

- P. Busia et al., ”EEGformer: Transformer-Based Epilepsy Detection on Raw EEG Traces for Low-Channel-Count Wearable Continuous Monitoring Devices,” 2022 IEEE Biomedical Circuits and Systems Conference (BioCAS), 2022, pp. 640-644, doi: 10.1109/BioCAS54905.2022.9948637.

Bibliography

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[2] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: (2015). arXiv: 1409.1556 [cs.CV].

[3] Christian Szegedy et al. "Rethinking the Inception Architecture for Computer Vision". In: *CoRR* arXiv:1512.00567v3 (2015). URL: https://arxiv.org/abs/1512.00567.

[4] Andy Brock et al. "High-Performance Large-Scale Image Recognition Without Normalization". In: *Proceedings of the 38th International Conference on Machine Learning*. Ed. by Marina Meila and Tong Zhang. Vol. 139. Proceedings of Machine Learning Research. PMLR, July 2021, pp. 1059–1071. URL: https://proceedings.mlr.press/v139/brock21a.html.

[5] Joseph Redmon et al. "You Only Look Once: Unified, Real-Time Object Detection". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 779–788. DOI: 10.1109/CVPR.2016.91.

[6] Christine Dewi et al. "Deep convolutional neural network for enhancing traffic sign recognition developed on Yolo V4". In: *Multimedia Tools and Applications* 81 (26 2022), pp. 37821–37845. DOI: 10.1007/s11042-022-12962-5. URL: https://doi.org/10.1007/s11042-022-12962-5.

[7] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation". In: *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*. Ed. by Nassir Navab et al. Cham: Springer International Publishing, 2015, pp. 234–241. ISBN: 978-3-319-24574-4.

[8] D. R. Sarvamangala and Raghavendra V Kulkarni. "Convolutional neural networks in medical image understanding: a survey". In: *Evolutionary Intelligence* 15 (1 2022), pp. 1–22. DOI: 10.1007/s12065-020-00540-3. URL: https://doi.org/10.1007/s12065-020-00540-3.

[9] Tursunov Anvarjon, Mustaqeem, and Soonil Kwon. "Deep-Net: A Lightweight CNN-Based Speech Emotion Recognition System Using Deep Frequency Features". In: *Sensors* 20.18 (2020). ISSN: 1424-8220. DOI: 10.3390/s20185212. URL: https://www.mdpi.com/1424-8220/20/18/5212.

[10] Fawziya M. Rammo and Mohammed N. Al-Hamdani. "Detecting The Speaker Language Using CNN Deep Learning Algorithm". In: *Iraqi Journal For Computer Science and Mathematics* 3.1 (Jan. 2022), pp. 43–52.

[11] Alwyn Burger et al. "An Embedded CNN Implementation for On-Device ECG Analysis". In: *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. 2020, pp. 1–6. DOI: 10.1109/PerComWorkshops48775.2020.9156260.

[12] Quan Liu et al. "Spectrum Analysis of EEG Signals Using CNN to Model Patient's Consciousness Level Based on Anesthesiologists' Experience". In: *IEEE Access* 7 (2019), pp. 53731–53742. DOI: 10.1109/ACCESS.2019.2912273.

[13] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[14] Paola Busia et al. "Target-Aware Neural Architecture Search and Deployment for Keyword Spotting". In: *IEEE Access* 10 (2022), pp. 40687–40700. DOI: 10.1109/ACCESS.2022.3166939.

[15] Paola Busia et al. "ALOHA: A Unified Platform-Aware Evaluation Method for CNNs Execution on Heterogeneous Systems at the Edge". In: *IEEE Access* 9 (2021), pp. 133289–133308. DOI: 10.1109/ACCESS.2021.3115243.

[16] Paola Busia et al. "EEGformer: Transformer-Based Epilepsy Detection on Raw EEG Traces for Low-Channel-Count Wearable Continuous Monitoring Devices". In: *2022 IEEE Biomedical Circuits and Systems Conference (BioCAS)*. 2022, pp. 640–644. DOI: 10.1109/BioCAS54905.2022.9948637.

[17] Di Liu et al. "Bringing AI To Edge: From Deep Learning's Perspective". In: 2020. arXiv: 2011.14808 [cs.LG].

[18] Ali Hassan Sodhro, Sandeep Pirbhulal, and Victor Hugo C. de Albuquerque. "Artificial Intelligence-Driven Mechanism for Edge Computing-Based Industrial Applications". In: *IEEE Transactions on Industrial Informatics* 15.7 (2019), pp. 4235–4243. DOI: 10.1109/TII.2019.2902878.

[19] Syed Umar Amin and M. Shamim Hossain. "Edge Intelligence and Internet of Things in Healthcare: A Survey". In: *IEEE Access* 9 (2021), pp. 45–59. DOI: 10.1109/ACCESS.2020.3045115.

[20] Tuomo Sipola et al. "Artificial Intelligence in the IoT Era: A Review of Edge AI Hardware and Software". In: *2022 31st Conference of Open Innovations Association (FRUCT)*. 2022, pp. 320–331. DOI: 10.23919/FRUCT54823.2022.9770931.

[21] Y. LeCun et al. "Deep learning". In: 2015.

[22] Google. *Accelerator Module datasheet*. 2022. URL: https://coral.ai/docs/module/datasheet/.

[23] NVIDIA. *NVDLA Primer*. 2022. URL: http://nvdla.org/primer.html.

[24] Movidius. *Movidius Neural Compute Stick: Accelerate deep learning development at the edge*. 2020. URL: https://developer.movidius.com/.

[25] Qualcomm. *Snapdragon 855+/860 Mobile Platform.* 2022. URL: https://www.qualcomm.com/products/application/smartphones/snapdragon-8-series-mobile-platforms/snapdragon-855-plus-and-860-mobile-platform.

[26] Nvidia Corporation. *Jetson TX2 platform technical specification.* URL: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-tx2/.

[27] ST Microelectronics. *Sensortile development kit.* 2022. URL: https://www.st.com/en/evaluation-tools/steval-stlkt01v1.html.

[28] Ambiq. *Apollo4 Ambiq.* June 2022. URL: https://ambiq.com/apollo4/.

[29] Greenwaves. *Gap9 processors for hearables and smart sensors.* Oct. 2022. URL: https://greenwaves-technologies.com/processor_noise_cancellation/.

[30] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks". In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)* (2016), pp. 367–379. DOI: https://doi.org/10.1109/ISCA.2016.40.

[31] Shuai Li et al. "A Novel FPGA Accelerator Design for Real-Time and Ultra-Low Power Deep Convolutional Neural Networks Compared With Titan X GPU". In: *IEEE Access* 8 (2020), pp. 105455–105471. DOI: 10.1109/ACCESS.2020.3000009.

[32] P. Meloni et al. "NEURAGHE : Exploiting CPU-FPGA Synergies for Efficient and Flexible CNN Inference Acceleration on Zynq SoCs". In: vol. 18. Dec. 2018. URL: https://doi.org/10.1145/3284357.

[33] NVIDIA. *cuDNN.* 2020. URL: https://developer.nvidia.com/cudnn.

[34] Liangzhen Lai, Naveen Suda, and Vikas Chandra. "CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs". In: *CoRR* abs/1908.09791 (2019). URL: https://arxiv.org/abs/1908.09791.

[35] Angelo Garofalo et al. "PULP-NN: A Computing Library for Quantized Neural Network inference at the edge on RISC-V Based Parallel Ultra Low Power Clusters". In: *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. 2019, pp. 33–36. DOI: 10.1109/ICECS46596.2019.8965067.

[36] Forrest Iandola and Kurt Keutzer. "Small Neural Nets Are Beautiful: Enabling Embedded Systems with Small Deep-Neural-Network Architectures". In: *Proceedings of the Twelfth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis Companion*. CODES '17. Seoul, Republic of Korea: Association for Computing Machinery, 2017. ISBN: 9781450351850. DOI: 10.1145/3125502.3125606. URL: https://doi.org/10.1145/3125502.3125606.

[37] Jia Deng et al. "Imagenet: A large-scale hierarchical image database". In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pp. 248–255.

[38] Andrew Howard et al. "Searching for MobileNetV3". In: *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. 2019, pp. 1314–1324. DOI: 10.1109/ICCV.2019.00140.

[39] Wei Fang, Lin Wang, and Peiming Ren. "Tinier-YOLO: A Real-Time Object Detection Method for Constrained Environments". In: vol. 8. 2020, pp. 1935–1944. DOI: 10.1109/ACCESS.2019.2961959.

[40] Tsung-Yi Lin et al. "Microsoft COCO: Common Objects in Context". In: (2014). Ed. by David Fleet et al., pp. 740–755.

[41] Song Han, Huizi Mao, and William J. Dally. "Deep Compression: Compressing Deep Neural Networks with Pruning Trained Quantization and Huffman Coding". In: International Conference on Learning Representations, 2016. arXiv: 1510.00149 [cs.CV].

[42] Amir Gholami et al. *A Survey of Quantization Methods for Efficient Neural Network Inference*. 2021. arXiv: 2103.13630 [cs.CV].

[43] Jungwook Choi et al. *PACT: Parameterized Clipping Activation for Quantized Neural Networks*. 2018. arXiv: 1805.06085 [cs.CV]. URL: https://arxiv.org/abs/1805.06085.

[44] Barret Zoph and Quoc Le. "Neural Architecture Search with Reinforcement Learning". In: *International Conference on Learning Representations*. 2017. URL: https://openreview.net/forum?id=r1Ue8Hcxg.

[45] Alex Krizhevsky. "Learning Multiple Layers of Features from Tiny Images". In: (2009). URL: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

[46] Esteban Real et al. "Regularized Evolution for Image Classifier Architecture Search". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 33.01 (July 2019), pp. 4780–4789. DOI: 10.1609/aaai.v33i01.33014780. URL: https://ojs.aaai.org/index.php/AAAI/article/view/4405.

[47] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. "Neural Architecture Search: A Survey". In: *Journal of Machine Learning Research* 20.55 (2019), pp. 1–21. URL: http://jmlr.org/papers/v20/18-598.html.

[48] Hadjer Benmeziane et al. "A Comprehensive Survey on Hardware-Aware Neural Architecture Search". In: (2021). URL: https://arxiv.org/abs/2101.09336.

[49] Mingxing Tan and Quoc V. Le. "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks". In: International Conference on Machine Learning, 2019. arXiv: 1905.11946 [cs.LG].

[50] L. Lai, N.Suda, and V. Chandra. "Not All Ops Are Created Equal!" In: *SysML*. 2018.

[51] M. Tan et al. "MnasNet: Platform-Aware Neural Architecture Search for Mobile". In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019, pp. 2815–2823. DOI: 10.1109/CVPR.2019.00293.

[52] Mark Sandler et al. "MobileNetV2: Inverted Residuals and Linear Bottlenecks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2018.

[53] Chi-Hung Hsu et al. "MONAS: Multi-Objective Neural Architecture Search using Reinforcement Learning". In: vol. abs/1806.10332. 2018.

[54] Weiwen Jiang et al. "Hardware/Software Co-Exploration of Neural Architectures". In: vol. 39. 2020, pp. 4805–4815.

[55] Qing Lu et al. "On Neural Architecture Search for Resource-Constrained Hardware Platforms". In: (). URL: `https://arxiv.org/abs/1911.00105`.

[56] Cong Hao et al. "FPGA/DNN Co-Design: An Efficient Design Methodology for IoT Intelligence on the Edge". In: New York, NY, USA: Association for Computing Machinery, 2019. ISBN: 9781450367257. DOI: `10.1145/3316781.3317829`. URL: `https://doi.org/10.1145/3316781.3317829`.

[57] X. Luo et al. "EdgeNAS: Discovering Efficient Neural Architectures for Edge Systems". In: *2020 IEEE 38th International Conference on Computer Design (ICCD)*. 2020, pp. 288–295. DOI: `10.1109/ICCD50377.2020.00056`.

[58] Han Cai, Ligeng Zhu, and Song Han. "ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware". In: *International Conference on Learning Representations*. 2019. URL: `https://arxiv.org/pdf/1812.00332.pdf`.

[59] Han Cai et al. "Once for All: Train One Network and Specialize it for Efficient Deployment". In: International Conference on Learning Representations, 2020. URL: `https://arxiv.org/pdf/1908.09791.pdf`.

[60] Hieu Pham et al. "Efficient Neural Architecture Search via Parameters Sharing". In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, July 2018, pp. 4095–4104. URL: `https://proceedings.mlr.press/v80/pham18a.html`.

[61] Yundong Zhang et al. "Hello Edge: Keyword Spotting on Microcontrollers". In: *CoRR* arXiv:1711.07128 (2017). URL: `https://arxiv.org/abs/1711.07128`.

[62] David Peter, Wolfgang Roth, and Franz Pernkopf. *End-to-end Keyword Spotting using Neural Architecture Search and Quantization*. 2021. arXiv: 2104.06666 [cs.SD]. URL: https://arxiv.org/abs/2104.06666.

[63] Colby Banbury et al. "MicroNets: Neural Network Architectures for Deploying TinyML Applications on Commodity Microcontrollers". In: (2020). URL: https://arxiv.org/abs/2010.11267.

[64] J. Czaja et al. "Applying the Roofline model for Deep Learning performance optimizations". In: vol. abs/2009.11224. 2020.

[65] Alberto Marchisio et al. "NASCaps: A Framework for Neural Architecture Search to Optimize the Accuracy and Hardware Efficiency of Convolutional Capsule Networks". In: New York, NY, USA: Association for Computing Machinery, 2020. ISBN: 9781450380263. DOI: 10.1145/3400302.3415731. URL: https://doi.org/10.1145/3400302.3415731.

[66] N. K. Jha and S. Mittal. "Modeling Data Reuse in Deep Neural Networks by Taking Data-Types into Cognizance". In: 2020, pp. 1–1. DOI: 10.1109/TC.2020.3015531.

[67] Lei Yang et al. "Co-Exploration of Neural Architectures and Heterogeneous ASIC Accelerator Designs Targeting Multiple Tasks". In: IEEE Press, 2020. ISBN: 9781450367257.

[68] B. Wu et al. "FBNet: Hardware-Aware Efficient ConvNet Design via Differentiable Neural Architecture Search". In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019, pp. 10726–10734. DOI: 10.1109/CVPR.2019.01099.

[69] G. Wu et al. "GPGPU performance and power estimation using machine learning". In: *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 2015, pp. 564–576. DOI: 10.1109/HPCA.2015.7056063.

[70] Chuan-Chi Wang et al. "PerfNet: Platform-Aware Performance Modeling for Deep Neural Networks". In: New York, NY, USA: Association for Computing Machinery, 2020. ISBN: 9781450380256. DOI: 10.1145/3400286.3418245. URL: https://doi.org/10.1145/3400286.3418245.

[71] A. Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[72] M. Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.* 2015. URL: http://tensorflow.org/.

[73] Md Zahangir Alom et al. "The History Began from AlexNet: A Comprehensive Survey on Deep Learning Approaches". In: CoRR, 2018. arXiv: 1803.01164 [cs.CV].

[74] S. Wang, A. Pathania, and T. Mitra. "Neural Network Inference on Mobile SoCs". In: 2020, pp. 1–1.

[75] S. Minakova, E. Tang, and T. Stefanov. "Combining Task- and Data-Level Parallelism for High-Throughput CNN Inference on Embedded CPUs-GPUs MPSoCs". In: *Embedded Computer Systems: Architectures, Modeling, and Simulation.* Springer International Publishing, 2020, pp. 18–35.

[76] C. Shea and T. Mohsenin. "Heterogeneous Scheduling of Deep Neural Networks for Low-Power Real-Time Designs". In: vol. 15. 4. Association for Computing Machinery, 2019. DOI: 10.1145/3358699.

[77] T. Do et al. "Real-Time Self-Driving Car Navigation Using Deep Neural Network". In: *2018 4th International Conference on Green Technology and Sustainable Development (GTSD).* 2018, pp. 7–12. DOI: 10.1109/GTSD.2018.8595590.

[78] C. Kyrkou et al. "DroNet: Efficient convolutional neural network detector for real-time UAV applications". In: *2018 Design, Automation Test in Europe Conference Exhibition (DATE).* 2018, pp. 967–972. DOI: 10.23919/DATE.2018.8342149.

[79] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: an Insightful Visual Performance Model for Floating Point Programs and Multicore Architectures". In: Communications of the ACM, 2009. URL: https://doi.org/10.1145/1498765.1498785.

[80] Hyoukjun Kwon et al. "Understanding Reuse, Performance, and Hardware Cost of DNN Dataflow: A Data-Centric Approach". In: New York, NY, USA: Association for Computing Machinery, 2019. ISBN: 9781450369381. DOI: 10.1145/3352460.3358252. URL: https://doi.org/10.1145/3352460.3358252.

[81] D. Justus et al. "Predicting the Computational Cost of Deep Learning Models". In: *2018 IEEE International Conference on Big Data (Big Data)*. 2018, pp. 3873–3882. DOI: 10.1109/BigData.2018.8622396.

[82] Andrew Anderson et al. "Performance-Oriented Neural Architecture Search". In: *2019 International Conference on High Performance Computing Simulation (HPCS)*. 2019, pp. 177–184. DOI: 10.1109/HPCS48598.2019.9188213.

[83] Tong Mo et al. "Neural Architecture Search for Keyword Spotting". In: *Proc. Interspeech 2020*. 2020, pp. 1982–1986. DOI: 10.21437/Interspeech.2020-3132.

[84] Guoguo Chen, Carolina Parada, and Georg Heigold. "Small-footprint keyword spotting using deep neural networks". In: *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2014, pp. 4087–4091. DOI: 10.1109/ICASSP.2014.6854370.

[85] T. Wang et al. "APQ: Joint search for network architecture, pruning and quantization policy". In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (2020), pp. 2075–2084. DOI: 10.1109/CVPR42600.2020.00215.

[86] A. Vaswani et al. "Attention is all you need". In: *Advances in Neural Information Processing Systems* 2017-December (2017), pp. 5999–6009.

[87] D. Alexey et al. "An image is worth 16x16 words: Transformers for image recognition at scale". In: *International Conference on Learning Representations* (2021).

[88] Axel Berg, Mark O'Connor, and Miguel Tairum Cruz. "Keyword Transformer: A Self-Attention Model for Keyword Spotting". In: *Proc. Interspeech 2021*. 2021, pp. 4249–4253. DOI: 10.21437/Interspeech.2021-1286.

[89] A. Burrello et al. “Bioformers: Embedding Transformers for Ultra-Low Power sEMG-based Gesture Recognition”. In: *IEEE 2022 DATE* (2022).

[90] Jiayao Sun, Jin Xie, and Huihui Zhou. “EEG Classification with Transformer-Based Models”. In: *2021 IEEE 3rd Global Conference on Life Sciences and Technologies (LifeTech)* (2021), pp. 92–93. DOI: 10.1109/LifeTech52111.2021.9391844.

[91] Zhe Wang et al. “Transformers for EEG-Based Emotion Recognition: A Hierarchical Spatial Information Learning Model”. In: *IEEE Sensors Journal* 22.5 (2022), pp. 4359–4368. DOI: 10.1109/JSEN.2022.3144317.

[92] J. Pedoeem et al. “TABS: Transformer Based Seizure Detection”. In: *2020 IEEE Signal Processing in Medicine and Biology Symposium (SPMB)* (2020), pp. 1–6. DOI: 10.1109/SPMB50085.2020.9353612.

[93] Steven Wyatt et al. “Environmental Sound Classification with Tiny Transformers in Noisy Edge Environments”. In: *2021 IEEE 7th World Forum on Internet of Things (WF-IoT)*. 2021, pp. 309–314. DOI: 10.1109/WF-IoT51360.2021.9596007.

[94] Sehoon Kim et al. “I-BERT: Integer-only BERT Quantization”. In: *Proceedings of the 38th International Conference on Machine Learning*. Ed. by Marina Meila and Tong Zhang. Vol. 139. Proceedings of Machine Learning Research. PMLR, July 2021, pp. 5506–5518. URL: https://proceedings.mlr.press/v139/kim21d.html.

[95] Pete Warden. “Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition”. In: *CoRR* arXiv:1804.03209 (2018). URL: https://arxiv.org/abs/1804.03209.

[96] P. Meloni et al. “ALOHA: an architectural-aware framework for deep learning at the edge”. In: *INTESA ’18: Proceedings of the Workshop on INTelligent Embedded Systems Architectures and Applications* (Oct. 2018), pp. 19–26. URL: https://doi.org/10.1145/3285017.3285019.

[97] Cristina Chesta and Luca Rinelli. “Modular approach to data preprocessing in ALOHA and application to a smart industry use case”. In: *CoRR* abs/2102.01349 (2021). arXiv: 2102.01349. URL: https://arxiv.org/abs/2102.01349.

[98] AI tools community. *Open Neural Network Exchange (ONNX).* Oct. 2021. URL: https://onnx.ai/.

[99] Paolo Meloni et al. "Optimization and Deployment of CNNs at the Edge: The ALOHA Experience". In: *Proceedings of the 16th ACM International Conference on Computing Frontiers.* CF '19. Alghero, Italy: Association for Computing Machinery, 2019, pp. 326–332. ISBN: 9781450366854. DOI: 10.1145/3310273.3323435. URL: https://doi.org/10.1145/3310273.3323435.

[100] Francesco Conti. *Technical Report: NEMO DNN Quantization for Deployment Model.* 2020. arXiv: 2004.05930 [cs.LG]. URL: https://arxiv.org/abs/2004.05930.

[101] A. Gordon et al. "MorphNet: Fast Simple Resource-Constrained Structure Learning of Deep Networks". In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition.* 2018, pp. 1586–1595. DOI: 10.1109/CVPR.2018.00171.

[102] L. L. Zhang et al. "Fast Hardware-Aware Neural Architecture Search". In: *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW).* 2020, pp. 2959–2967. DOI: 10.1109/CVPRW50498.2020.00354.

[103] Hang Qi, R. Sparks Evan, and Ameet Talwalkar. "PALEO: a Performance Model for Deep Neural Networks". In: *International Conference on Learning Representations, 2017.* 2017.

[104] NVIDIA Corporation. *CUDA C++ Best Practices Guide.* https://docs.nvidia.com/cuda/pdf/CUDACBestPracticesGuide.pdf. 2021.

[105] Xilinx. *Xilinx Power Estimator.* URL: https://www.xilinx.com/products/technology/power/xpe.html.

[106] Karthik Chandrasekar et al. *DRAMPower: Open-source DRAM Power & Energy Estimation Tool.* URL: http://www.drampower.info.

[107] Nvidia Corporation. *NVIDIA TensorRT Deep learning library and inference optimizer.* URL: https://developer.nvidia.com/tensorrt.

[108] Paolo Meloni et al. “Exploring NEURAghe: A Customizable Template for APSoC-Based CNN Inference at the Edge”. In: *IEEE Embedded Systems Letters* 12.2 (2020), pp. 62–65. DOI: 10.1109/LES.2019.2947312.

[109] Intel. *AN531: Reducing Power with Hardware Accelerators*. 2008.

[110] Emanuele Dilettoso, Santi Rizzo, and Nunzio Salerno. “A Weakly Pareto Compliant Quality Indicator”. In: vol. 2017. Mar. 2017. DOI: 10.3390/mca22010025.

[111] Eckart Zitzler, Kalyanmoy Deb, and Lothar Thiele. “Comparison of Multiobjective Evolutionary Algorithms: Empirical Results”. In: vol. 8. 2. Cambridge, MA, USA: MIT Press, 2000. DOI: 10.1162/106365600568202. URL: https://doi.org/10.1162/106365600568202.

[112] A. L. Goldberger et al. “Physiobank, physiotoolkit, and physionet: components of a new research resource for complex physiologic signals”. In: *circulation* 101 (2000), e215–e220.

[113] A. H. Shoeb. “Application of machine learning to epileptic seizure onset detection and treatment”. In: *Ph.D. dissertation, MIT* (2009).

[114] World Health Organization. *Epilepsy*. June 2022. URL: https://www.who.int/news-room/fact-sheets/detail/epilepsy.

[115] Thorir Mar Ingolfsson et al. “Towards Long-term Non-invasive Monitoring for Epilepsy via Wearable EEG Devices”. In: *BioCAS 2021 - IEEE Biomedical Circuits and Systems Conference, Proceedings* (2021). DOI: 10.1109/BioCAS49922.2021.9644949.

[116] Dionisije Sopic, Amir Aminifar, and David Atienza. “e-Glass: A Wearable System for Real-Time Detection of Epileptic Seizures”. In: *2018 IEEE International Symposium on Circuits and Systems (ISCAS)* (2018), pp. 1–5. DOI: 10.1109/ISCAS.2018.8351728.

[117] Nhat Pham et al. “WAKE: A behind-the-Ear Wearable System for Microsleep Detection”. In: *Association for Computing Machinery*. MobiSys ’20 (2020), pp. 404–418. DOI: 10.1145/3386901.3389032. URL: https://doi.org/10.1145/3386901.3389032.

[118] Turkey N. Alotaiby et al. "EEG seizure detection and prediction algorithms: a survey". In: *EURASIP Journal on Advances in Signal Processing* (2014). DOI: 10.1186/1687-6180-2014-183. URL: https://doi.org/10.1186/1687-6180-2014-183.

[119] Paul Yash. "Various epileptic seizure detection techniques using biomedical signals: a review". In: *Brain Informatics* (2018). DOI: 10.1186/s40708-018-0084-z. URL: https://doi.org/10.1186/s40708-018-0084-z.

[120] J. Prasanna et al. "Automated Epileptic Seizure Detection in Pediatric Subjects of CHB-MIT EEG Database-A Survey." In: *J Pers Med.* (2021). DOI: 10.3390/jpm11101028.

[121] Alex Van Esbroeck et al. "Multi-task seizure detection: addressing intra-patient variation in seizure morphologies". In: *Machine Learning* 102.3 (2016), pp. 309–321. DOI: 10.1007/s10994-015-5519-7. URL: https://doi.org/10.1007/s10994-015-5519-7.

[122] Chen Guangyi et al. "Automatic Epileptic Seizure Detection in EEG Using Nonsubsampled Wavelet–Fourier Features". In: *Journal of Medical and Biological Engineering* (2017), pp. 123–131. DOI: 10.1007/s40846-016-0214-0. URL: https://doi.org/10.1007/s40846-016-0214-0.

[123] S. M. Shafiul Alam and M. I. H. Bhuiyan. "Detection of Seizure and Epilepsy Using Higher Order Statistics in the EMD Domain". In: *IEEE Journal of Biomedical and Health Informatics* 17.2 (2013), pp. 312–318. DOI: 10.1109/JBHI.2012.2237409.

[124] Abhijeet Bhattacharya, Tanmay Baweja, and S. P. K. Karri. "Epileptic Seizure Prediction Using Deep Transformer Model". In: *International Journal of Neural Systems* 32.02 (2022). DOI: https://doi.org/10.1142/S0129065721500581.

[125] Jianzhuo Yan et al. "Seizure Prediction Based on Transformer Using Scalp Electroencephalogram". In: *Applied Sciences* 12.9 (2022). ISSN: 2076-3417. DOI: 10.3390/app12094158. URL: https://www.mdpi.com/2076-3417/12/9/4158.

[126] Chaosong Li et al. "Seizure Onset Detection Using Empirical Mode Decomposition and Common Spatial Pattern". In: *IEEE Transactions on Neural Systems and Rehabilitation Engineering* 29 (2021), pp. 458–467. DOI: 10.1109/TNSRE.2021.3055276.

[127] Mingyang Li, Xiaoying Sun, and Wanzhong Chen. "Patient-specific seizure detection method using nonlinear mode decomposition for long-term EEG signals". In: *Medical & Biological Engineering & Computing* 58 (12 Dec. 2020). URL: https://doi.org/10.1007/s11517-020-02279-6.

[128] Xiaoshuang Wang et al. "One dimensional convolutional neural networks for seizure onset detection using long-term scalp and intracranial EEG". In: *Neurocomputing* 459 (2021), pp. 212–222. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2021.06.048.

[129] Thorir Mar Ingolfsson et al. "Energy-Efficient Tree-Based EEG Artifact Detection". In: *2022 44th Annual International Conference of the IEEE Engineering in Medicine Biology Society (EMBC)*. 2022, pp. 3723–3728. DOI: 10.1109/EMBC48229.2022.9871413.

[130] Muhammad Awais Bin Altaf and Jerald Yoo. "A 1.83 μJ/Classification, 8-Channel, Patient-Specific Epileptic Seizure Classification SoC Using a Non-Linear Support Vector Machine". In: *IEEE Transactions on Biomedical Circuits and Systems* 10.1 (2016), pp. 49–60. DOI: 10.1109/TBCAS.2014.2386891.

[131] Muhammad Awais Bin Altaf, Chen Zhang, and Jerald Yoo. "A 16-Channel Patient-Specific Seizure Onset and Termination Detection SoC With Impedance-Adaptive Transcranial Electrical Stimulator". In: *IEEE Journal of Solid-State Circuits* 50.11 (2015), pp. 2728–2740. DOI: 10.1109/JSSC.2015.2482498.

[132] Matteo Spallanzani et al. *QuantLab: a Modular Framework for Training and Deploying Mixed-Precision NNs*. https://cms.tinyml.org/wp-content/uploads/talks2022/Spallanzani-Matteo-Hardware.pdf. Mar. 2022.

[133] Alessio Burrello et al. "A Microcontroller is All You Need: Enabling Transformer Execution on Low-Power IoT Endnodes". In: *2021 IEEE International Conference on Omni-Layer Intelligent Systems (COINS)* (2021), pp. 1–6. DOI: 10.1109/COINS51742.2021.9524173.

[134] Greenwaves. *Ultra Low Power GAP Processors.* Oct. 2022. URL: https://greenwaves-technologies.com/low-power-processor/.