Universiteit Leiden
The Netherlands

ALOHA: a unified platform-aware evaluation method for CNNs execution on heterogeneous systems at the edge

Busia, P.; Minakova, S.; Stefanov, T.P.; Raffo, L.; Meloni, P.

Citation

Busia, P., Minakova, S., Stefanov, T. P., Raffo, L., & Meloni, P. (2021). ALOHA: a unified platform-aware evaluation method for CNNs execution on heterogeneous systems at the edge. *Ieee Access*, *9*, 133289-133308. doi:10.1109/ACCESS.2021.3115243

Version:	Publisher's Version
License:	Creative Commons CC BY 4.0 license
Downloaded from:	https://hdl.handle.net/1887/3273755

Note: To cite this publication please use the final published version (if applicable).

IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received July 23, 2021, accepted September 13, 2021, date of publication September 23, 2021, date of current version October 5, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3115243

ALOHA: A Unified Platform-Aware Evaluation Method for CNNs Execution on Heterogeneous Systems at the Edge

PAOLA BUSIA[1], SVETLANA MINAKOVA[2], TODOR STEFANOV[2], (Member, IEEE), LUIGI RAFFO[1], AND PAOLO MELONI[1], (Member, IEEE)

[1]DIEE, University of Cagliari, 09123 Cagliari, Italy
[2]LIACS, Leiden University, 2333 CA Leiden, The Netherlands

Corresponding author: Paolo Meloni (paolo.meloni@unica.it)

This work was supported by the European Union's Horizon 2020 Research and Innovation Program under Grant 780788.

ABSTRACT CNN design and deployment on embedded edge-processing systems is an error-prone and effort-hungry process, that poses the need for accurate and effective automated assisting tools. In such tools, pre-evaluating the platform-aware CNN metrics such as latency, energy cost, and throughput is a key requirement for successfully reaching the implementation goals imposed by use-case constraints. Especially when more complex parallel and heterogeneous computing platforms are considered, currently utilized estimation methods are inaccurate or require a lot of characterization experiments and efforts. In this paper, we propose an alternative method, designed to be flexible, easy to use, and accurate at the same time. Considering a modular platform and execution model that adequately describes the details of the platform and the scheduling of different CNN operators on different platform processing elements, our method captures precisely operations and data transfers and their deployment on computing and communication resources, significantly improving the evaluation accuracy. We have tested our method on more than 2000 CNN layers, targeting an FPGA-based accelerator and a GPU platform as reference example architectures. Results have shown that our evaluation method increases the estimation precision by up to 5× for execution time, and by 2× for energy, compared to other widely used analytical methods. Moreover, we assessed the impact of the improved platform-awareness on a set of neural architecture search experiments, targeting both hardware platforms, and enforcing 2 sets of latency constraints, performing 5 trials on each search space, for a total number of 20 experiments. The predictability is improved by 4×, reaching, with respect to alternatives, selection results clearly more similar to those obtained with on-hardware measurements.

INDEX TERMS Convolutional neural networks, edge-computing, platform awareness.

I. INTRODUCTION

Convolutional Neural Networks (CNNs) [1] are biologically inspired graph computational models, characterized by a large number of parameters and a high degree of parallelism. Due to their ability to handle large, unstructured data, CNNs are widely used to perform tasks such as image and video recognition, image segmentation, natural language processing, and many others [2]. Nowadays, CNNs are the backbone of many applications, such as navigation in self-driving cars [3], medical image recognition [4], surveillance [5], and others [2]. Due to the intense computation workload associated with their execution, CNNs often require, especially when operating at the edge, to exploit acceleration on dedicated processing elements, usually heterogeneous and highly parallel. CNN inference has been ported on a wide spectrum of platforms: from high-performance GPU clusters to embedded systems and mobile devices [6], [7]. Nevertheless, the landscape of CNN-enabling cores and processors in literature is increasingly vast: the majority of silicon vendors and market actors are proposing new accelerator or processor architectures designed to improve the efficiency of CNN execution ([8]–[12]). Rounding up the numbers, the main three classes of processing elements exploited for this kind

The associate editor coordinating the review of this manuscript and approving it for publication was Fan Zhang.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

of workload are CPUs, GPUs, and dedicated processing elements. Understanding the execution of a specific CNN architecture on such complex processing systems, before the actual deployment, is a key need during several design steps: e.g. during target platform selection, CNN topology definition also referred to as Neural Architecture Search (NAS), task-to-core mapping optimization, code-level optimization. Most of these steps are time-consuming activities requiring sufficient expertise in the field of Deep Learning (DL) [2], to be performed manually.

Thus, automated design flows and tools are appearing in the literature, to assist non-experts in such challenging tasks.

However, most tools reported in the literature have a limited degree of platform awareness: they fail to capture the effect of potential design choices on the performance metrics achievable by a CNN architecture under consideration executed on a target computing platform, especially when dealing with more complex processing systems, endowed with accelerators, highly-parallel processors and/or GPUs. Estimation methods implemented in these tools are inaccurate ([13]–[15]), or not sufficiently general ([16]–[23]), or require a lot of design experiments and modeling skills to be used ([24]–[26]).

A common unified method that solves all these issues, implementing platform-awareness within automated tools for CNN design, is still missing.

Therefore, in this paper, we propose the ALOHA[1] method for the evaluation of platform-dependent metrics of a CNN, executed on a heterogeneous platform. Our method relies on a platform-aware evaluation model, described in Section VI, designed to:

- **provide realistic and accurate results:** the model is capable of capturing platform-aware characteristics, such as occupancy of platform processors, exploitation of parallelism available in a platform by CNN operators, repeated data transfers occurring during CNN execution, and others;
- **be flexible:** the model is not dependent on any specific processing element architectural template. Characteristics that are captured in the model are abstract enough to be usable for the description of significantly different platform organizations and structures;
- **be modular:** one component of the model describes the platform, while a second part describes the deployment strategy which is used by the implementation of CNN layers (defined by the user or the selected library). This improves both accuracy and re-usability because both components can be adopted in different design cases.
- **require low development effort:** the model does not require benchmarking. All the information required to capture the platform and library can be easily derived from specs or a general understanding of the platform/library operation principles.

To evaluate our method, we compare the accuracy provided by our proposed method with others with a similar level of abstraction and development effort, considering two heterogeneous platforms as a reference: an open-source FPGA-based platform called NEURAghe [27] and a GPU-based Jetson TX2 platform [28]. The architectures presented in this paper are exactly chosen to represent the three classes of common processing elements in the embedded domain. JetsonTX2 is a SoC integrating CPU and GPU. NEURAghe is implemented on a SoC that integrates CPU and a CNN-dedicated processing element implemented on reconfigurable logic. Thus we believe that overall, this selection covers most of the embedded landscape.

The comparison shows that our approach significantly improves the evaluation precision. Moreover, we perform several NAS experiments, optimizing the topology of a CNN to perform classification on the CIFAR-10 dataset [29], under user-defined latency constraints and targeting the aforementioned platforms. For each exploration, we used different kinds of evaluation methods to confront candidate design points with the constraint. Comparing the NAS results obtained using our method with those obtained using other comparable models, we show that our method significantly improves predictability, bringing NAS selection very similar to the one obtained by actual on-hardware measurements.

A. PAPER CONTRIBUTIONS

The main novel contributions in this paper can be summarized as:

- an accurate, easy to create and yet generalizable and reusable platform model and evaluation method, proposed in Section V and Section VI, suitable to implement platform-awareness in CNN design and optimization tools;
- assessment of the impact of platform-awareness on the latency estimation (Section VIII-A), reducing by 3× to 5× the average error in CNN latency estimation, compared to commonly used methods such as the Roofline model [16] and operation count, for layer-level evaluation, and by 1.6× when considering aggregated CNN-level results on multiple cores (Section VIII-D);
- assessment of the impact of platform-awareness on the energy estimation (Section VIII-B), showing a 1.9× estimation precision improvement;
- assessment of the impact of platform-awareness on NAS (Section VIII-C), reducing thanks to the proposed method the latency and accuracy deviation from a similar NAS exploration having access to actual on-hardware measurements by a factor of 4, compared to the alternative methods examined.

II. RELATED WORK

As an answer to the demand for CNN-based edge-processing, custom-developed devices and computing systems, an

[1]The ALOHA project is available at https://www.aloha-h2020.eu/ and aims at developing a framework providing several tools for architecture-aware CNN exploration. This work only focuses on modeling, and it does not deal with adaptivity, pruning, and quantization themes.

TABLE 1. Comparison of methods for evaluation of platform-aware CNN metrics.

Eval. method category	Methods	Metric	Accuracy	Re-usability	Modularity
OPS	[13]	latency	low	very high	X
	[14]	memory, latency, energy			
Roofline	[30]	latency	low/ medium	high	X
Specialized analytical methods	[23]	latency, energy, memory	medium/ high	low	X
	[21], [31]	latency, energy			
	[19], [20], [22]	latency, energy, throughput		medium	
Measurements	[32]	latency	high	very low	X
	[33]	latency, energy			
Look-up tables (LUT)	[34]	latency	high	low	X
	[35]	latency	high	medium/ low	
ML	[36]	latency, energy	high	very low	X
	[26]	latency			
ALOHA	**this work**	**latency, energy, throughput**	**medium/ high**	**high**	✓

ever-increasing number of automated/assisted design tools have been recently proposed. Among such tools, some act very early in the design flow, when the processing platform and the on-platform deployment strategy are still not already selected or physically available. A key example of such early intervention is provided by Neural Architecture Search (NAS). Multi-objective NAS has been an active research topic during the last several years, and a large number of methods, capable of evaluating platform-dependent CNN metrics to assist it, have been proposed. Table 1 provides the overview and comparison of these methods, summarized into categories, listed in Column 1. Every evaluation method is supplied with a list of evaluated platform-aware metrics (Column 3). Every evaluation methods category is supplied with a list of methods belonging to it (Column 2), and characterized with: 1) the method accuracy (Column 4); 2) the method re-usability (Column 5); 3) modularity (Column 6). The method re-usability determines how sensitive the evaluation method category is to a specific CNN architecture or/and hardware platform, and determines the applicability of the evaluation method category. For example, the OPS category, shown in Row 2 of Table 1, has very high re-usability: it can be easily applied to a wide range of CNNs and hardware platforms and does not require any modifications if the range of explored CNNs or target platform is changed. In contrast to the methods from the OPS category, the methods, based on ML models, and shown in Rows 12 to 13 of Table 1, demonstrate very low re-usability. Once designed, the ML models, used in these methods, are only applicable to a specific set of explored CNNs and a specific target platform. If the target platform or set of explored CNNs changes, the ML-based models have to be designed from scratch. Low re-usability of an evaluation method might limit the use of this method or involve large design time overheads. With the rapidly increasing number and diversity of CNNs as well as platforms, used to execute the CNNs, high evaluation method re-usability is an important quality metric of the evaluation methods, used in NAS.

Modularity (Column 6) specifies whether an evaluation method accounts for the modular composition of the platform, considering the distribution of CNN layers over the processors of a heterogeneous platform as well as for a specific schedule, associated with the CNN execution. Typically the execution of a CNN, on an accelerator-based platform, involves layer-by-layer execution of a CNN and offloading of computations within every CNN layer onto a platform accelerator [2]. Such CNN execution is typical for the majority of widely used DL frameworks such as PyTorch [37] or TensorFlow [38]. However, the ongoing research in the field of Edge AI is exploring alternative, more efficient ways to execute CNNs on heterogeneous edge platforms [7]. For example, methods, proposed in papers [39]–[41], enable for better utilization of computational resources, available on the platform. The exploitation of these methods can significantly affect platform-aware CNN metrics, such as CNN throughput and energy consumption. Thus, for efficient evaluation of platform-aware metrics of a CNN, executed at the edge, evaluation methods should have means to account for such advanced CNN execution methods.

The OPS-based evaluation methods, shown in Rows 2 to 3 of Table 1, estimate the CNN latency using the number of operations (OPs) required to execute a CNN. Such evaluation methods are simple to use and are characterized by high re-usability. However, the predictions provided by the ops-based evaluation methods are often inaccurate [42], [43]. Low accuracy in the evaluation methods might lead to a large margin between predicted CNN metric and real CNN metric measured when a CNN is executed on the target platform. Such a large margin is unacceptable for the design of CNN-based applications with strict resource constraints, such as self-driving cars [3] or object recognition on drones [44]. Unlike the OPS-based methods, our method provides more accurate evaluations, and thus, allows us to obtain more realistic predictions of platform-dependent CNN metrics.

The Roofline methods, shown in Row 4 of Table 1, evaluate platform-dependent metrics of a CNN, using the analytical platform-aware Roofline model [16]. These methods, in addition to the number of OPS performed by a CNN during its execution, take into account the impact of memory access on the platform-aware CNN metrics, which allows these

methods to perform more precise CNN metrics evaluation, compared to the OPS-based methods, explained above. However, as the OPS-based evaluation methods, the Roofline methods lack evaluation accuracy. In our method, we propose a novel platform-aware evaluation model, alternative to the Roofline model. Our evaluation model considers a wider range of platform-aware characteristics, and provides a more precise evaluation of platform-dependent CNN metrics, compared to the Roofline model.

Specialized analytical methods, shown in Rows 5 to 7 of Table 1, use highly detailed representations of hardware platforms to provide a precise estimation of platform-dependent CNN metrics. However, the utilization of highly specialized hardware specifications leads to narrow application and low re-usability of these methods. For example, the authors of [19]–[21] target the exploitation of a precise roofline-based model for FPGA codesign. However, the model, utilized in [19]–[21] cannot be applied to other platforms, such as CPUs-GPUs platforms. Analogously, the work in [22] explores ASICs codesign, through performance evaluation based on MAESTRO [45], which makes the evaluation method, proposed in [22], only applicable to ASICs-based platforms. With the rapidly increasing number and diversity of devices, used to execute CNNs, such high specialization significantly limits the use of these methods. In contrast to these methods, we propose an abstract high-level specification of a hardware platform, which contains many platform-aware details, affecting platform-dependent metrics of CNNs, and which applies to a wide range of diverse hardware platforms. Thus, our method enables for higher applicability and re-usability, compared to the methods proposed in [19]–[21].

The measurement-based methods, Rows 8-9 of Table 1, are based on actual measures of real latency of CNNs on the target platforms. Some similar approaches, Rows 10-11 of Table 1, use measured latency values for CNN components collected in Look-up Tables (LUTs) to produce by composition the estimation of entire CNNs. These methods ensure highly precise evaluations. However, they involve a large number of measurements of specific CNNs mapped on a specific target platform, especially in the case of measurement-based methods [32], [33], or limit the range of analyzable CNNs to those composed by modules available in the LUT. Thus, these methods show very low re-usability. On the contrary, our method uses abstract platform specification and CNN description, enabling its high re-usability.

The evaluation methods, based on ML models, and shown in Rows 12 to 13 of Table 1, use trainable ML models such as neural networks or regression models, to predict the platform-dependent metrics of a CNN. However, like the measurement-based methods and LUTs, explained above, the ML methods require a large amount of platform- and CNN-specific measurements, and demonstrate very low re-usability. Unlike these methods, ours does not require platform- and CNN-specific measurements and demonstrates high re-usability.

TABLE 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}.

Method	Data Collection time $n_sample * t * N_{avg}$	Training time
[25]	$75000 * t * 5$	(300 epochs)
[26]	$80000 * t * N_{avg}$	1h (1000 epochs)
[32]	$90000 * t * 50$	20min (150 epochs)
[36]	$447 * 108 * t * N_{avg}$	not specified

Table 2 provides an overview of the required profiling time for the highly accurate estimation methods based on ML models. All of the listed works require a significant amount of deployments and measurements: assuming the execution time of a network to be, on average, equal to 15 ms, for example, [25] would require over 1 hour of data collection, while over 18 hours are needed in [32]. This is a very soft hypothesis, as in [26] almost 2 weeks of data collection time are claimed. In some cases, the training procedure can be exploited for a wide range of targets (e.g. [36] evaluates 447 different GPU configurations, while [25] suggests training a single network for predicting performance on multiple hardware), while, in general, such procedure has to be repeated for each target platform. On the contrary, the ALOHA method does not require benchmarking.

Finally, to the best of our knowledge, our method advances the state-of-the-art combining the flexibility characteristics of analytical methods with improved accuracy. Thus, it provides results similar to measurement-based methods, without requiring intensive modeling effort. Moreover, to the best of our knowledge, this is the first work that implements modular pre-estimation, taking into account the mapping and concurrent execution of different computational kernels on different processing cores, among those available on the platform, and providing system-level performance estimation.

III. BACKGROUND

We summarize in this section some background notions that will be widely used in the paper. In Section III-A we describe the specifics of CNN architectures. In Section III-B and III-C we briefly present the theory of the analytical methods that we compare our method with in the following sections, the OPS count and the Roofline models.

A. CNN COMPUTATIONAL KERNEL DESCRIPTION

A Convolutional Neural Network (CNN) can be represented as a directed acyclic computational graph $CNN(L, E)$ with a set of nodes L, also called layers, and a set of edges E [2]. An example of a CNN with layers $L = \{l_1, l_2, l_3\}$ and edges $E = \{e_{12}, e_{23}\}$ is given in Figure 1.

Every layer $l_i \in L$ represents a part of CNN functionality. It accepts as input data X_i, provided by other CNN layers or external sources, and provides as output data Y_i. The

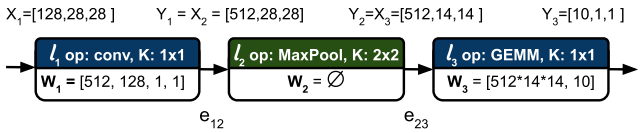


FIGURE 1. CNN.

layers input and output data are stored in multidimensional arrays, called *tensors* [2]. In this paper, the input and output data tensors have the format $[T.B, T.C, T.H, T.W]$, where T denotes the tensor; $T.B$, $T.C$, $T.H$, $T.W$ are the batch size, the number of channels, the height and the width of the tensor, respectively. Being one of the most common processing choices for embedded inference execution, and the one required by most real-time applications, we place particular focus on the processing case where the batch size is equal to 1, where $T.B$ dimension can be omitted in the notation. To obtain the output data Y_i from the input data X_i, layer l_i moves along its input with a sliding window K_i, and applies a CNN operator op_i (such as convolution, MaxPooling, GEMM, RelU etc. [2]), parameterized with weights W_i to its input data tensor X_i. We consider layer l_1 in the CNN structure in Figure 1 as an example of CNN layer, with input data tensor $X_1 = [128, 28, 28]$, output data tensor $Y_1 = [512, 28, 28]$, and a 2-dimensional sliding window K_1 with width $K_1.W = 1$, and height $K_1.H = 1$.

Each CNN edge $e_{ij} \in E$ specifies a data dependency between CNN layers l_i and l_j, such that layer l_j accepts as input the data tensor Y_i, produced by layer l_i. The data dependencies, specified by the CNN edges, determine the order, in which CNN layers are executed on a target platform. Typically, CNN layers are executed in sequential order, i.e., a CNN execution can be represented as $|L|$ computational steps, where at every i-th computational step, CNN layer $l_i \in L$ is executed.

Considering the memory footprint of current state-of-the-art CNNs and the on-chip memory available in most embedded processing platforms, execution of layer l_i on a target platform typically involves:

1) loading of input data X_i and weights W_i of layer l_i from the global memory of the platform into the local memories of the platform processor, allocated for execution of l_i;
2) execution of the computations within the layer on the allocated platform processor;
3) copying of output data Y_i of layer l_i from the local memories of the allocated processor into the platform global memory;

The computations, performed within every CNN layer, are data-parallel computations, that can be represented using a set of nested loops, bound by the dimensions of tensors X_i and Y_i, as well as sliding window K_i. The nested loops enclose a simple operation, applied to the input and output data of the CNN layer. Hereinafter, we refer to such a set of nested loops as to the *computational tensor* of a CNN layer.

TABLE 3. Layer-specific computational tensor parameter.

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

```
for batch in range(BS):
 for o_feat in range(OF):
  for i_feat in range(IF):
   for fh in range(FH):
    for fw in range(FW):
     for k_y in range(KH):
      for k_x in range(KW):
       do simple_op
```

Listing 1. Generic CNN layer computational tensor.

```
for o_feat in range(512):
 for i_feat in range(128):
   for fh in range(28):
    for fw in range(28):
     for k_y in range(1):
      for k_x in range(1):
       do MAC
```

Listing 2. Computational tensor of convolutional layer l_1.

In this paper, we represent the computational tensor of every CNN layer, using a generic computational tensor and Table 3. The generic computational tensor, given in Listing 1, represents computations within every layer of a CNN architecture as a set of 7 loops, bound by generic loop bounds BS, OF, IF, FH, FW, KH and KW, and enclosing a generic simple operation *simple_op*. To represent computations within a specific CNN layer l, the loop bounds and the simple operation in the generic computational tensor are replaced with their respective layer-specific values. For example, to represent the computations within CNN layer l_1, shown in Figure 1, the generic loop bounds OF, IF, FH, FW, KH, KW of the generic computational tensor in Listing 1, are replaced in Listing 2 with their layer-specific values: $Y_1.C = 512$, and $X_1.C = 128$, also called number of output and input features of the layer, respectively; $Y_1.H = 28$, and $X_1 = 28$, the height and width of the layer output data tensor; $K_1.H = 1$, and $K_1.W = 1$, the height and width of the layer sliding window K_1; and the generic operation *simple_op* is replaced by the layer-specific simple operation MAC. The external loop on batch size BS is omitted, assuming batch size equal to 1.

Table 3 specifies how the generic loop bounds and generic simple operations are replaced with their layer-specific values for CNN layers, performing various CNN operators. In Table 3, Column 1 lists common CNN operators; Columns 2 to 8 show how generic loop bounds BS, OF, IF, FH, FW, KH and KW of the generic computational tensor, given in Listing 1, are replaced by the dimensions of input layer X, output layer Y and sliding window K of a CNN layer; Column 9 shows how generic operator *simple_op* of the generic computational tensor is replaced with a layer-specific

simple operator for a CNN layer. We note, that if needed, Table 3 can be customized or extended with new CNN operators or their specific implementations.

B. OPS-BASED PERFORMANCE PREDICTION

Most approaches in the literature evaluate the execution latency of a CNN as:

$$t_{OPS} = OPS/AP \tag{1}$$

where the attainable performance AP is considered equal to the peak performance of the hardware platform, $APmax$ [OPS/s], while OPS is the total number of operations, that must be executed during the CNN inference. The value of OPS is computed as:

$$OPS = \prod_{n=1}^{N} T.dim_n * \#OPS_enclosed \tag{2}$$

where $\prod_{n=1}^{N} T.dim_n$ is a product of all computational tensor dimensions dim_n, $n \in [1, N]$; $\#OPS_enclosed$ is the number of OPS in a simple operation, enclosed in the loops of the layer computational tensor. For example, the total number of operations, performed by the convolutional layer, represented as a computational tensor in Listing 2, 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 all computational tensor dimensions and 2 indicates that every MAC operation, enclosed in the loops of the computational tensor, consists of two operations: one multiplication and one addition.

C. ROOFLINE MODEL

The well-known Roofline Model [16] takes into account the impact of memory access on execution time. It exists in the OPS/s vs OPS/byte plane, and combines peak performance, represented as a horizontal line, with the actual bandwidth available to off-chip memory, defining a diagonal line with 45° inclination. The best-case execution time for a given kernel is defined by the operating point in the roofline. This is obtained as the intersection with the vertical line representing the kernel's Operational Intensity, $Int(l_i)$, defined as the ratio among OPS count and total data transferred:

$$Int(l_i) = OPS(l_i)/Traffic_{mem}(l_i) \tag{3}$$

For a CNN layer l_i, explained in Section III-A, the amount of data transfers, performed during the layer execution, can be estimated as:

$$Traffic_{mem}(l_i) = Size(X_i) + Size(W_i) + Size(Y_i) \tag{4}$$

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:

$$Size(T) = \prod_{n=1}^{N} T.dim_n * sizeof(pixel) \tag{5}$$

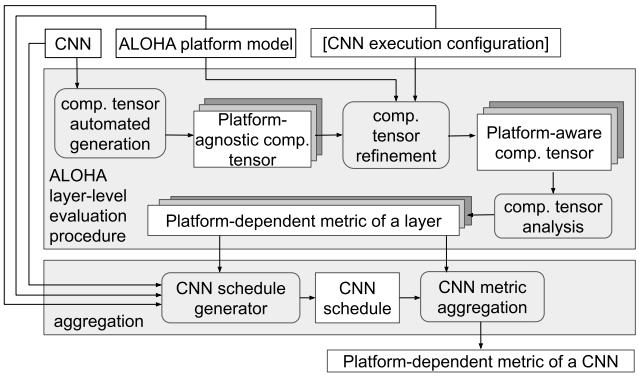


FIGURE 2. ALOHA methodology design flow.

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.

Based on this representation, execution time is estimated according to Equation 1, with $AP = AP_{roof}$ evaluated as:

$$AP_{roof} = min(AP_{max}, Int * bw) \tag{6}$$

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

IV. ALOHA METHOD

In this section, we propose our ALOHA method. The main purpose of our method is to evaluate platform-dependent metrics, such as latency and energy consumption, related to the execution of a CNN on a specific accelerator-based target platform. The design flow of our method[2] is shown in Figure 2. Our method accepts as inputs:

- a CNN description, for example, in ONNX format [46], which describes a CNN as a directed computational graph, explained in Section III-A;
- a specification of the accelerator-based platform, represented using the novel ALOHA platform model, proposed in Section V;
- (optionally) a CNN execution configuration, explained in details in Section VII.

Such input is used through the phases of the proposed method:

- Phase 1: The algorithm architecture and the platform-aware characteristics, specified in the platform model, are used by the ALOHA layer-level evaluation procedure, proposed in Section VI. The procedure is itself composed of several steps:
  - first, as described in Section VI-A, the procedure generates a computational tensor for every layer in the CNN, depending on the layer features; we indicate this process as *Computational tensor generation*;
  - second, as described in Section VI-B, the computational tensor is annotated considering platform features: the loops of operations and data transfers

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

are ordered, mapped on available parallel hardware operators, and partitioned 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*;
  - third, as described in Section VI-C, the obtained tensor is analyzed to derive accurate estimation of the metrics under evaluation; this process is referred to as *Computational tensor analysis.*
- Phase 2: The obtained layer-level estimations are passed as input to an *aggregation* module, described in Section VII, that schedules the execution of the different components on the processing elements in the platform, to deliver the final estimation of a platform-dependent metric of a CNN.

V. ALOHA PLATFORM MODEL

When it comes to describing details of a platform, simple specifications, such as the Roofline model, fail to describe some characteristics of the implementation, or the execution dataflow, that may have a significant impact on the platform-aware metrics of a CNN. Therefore, we have defined an abstract and generalizable, yet more detailed model which enables us to capture such characteristics. In this section, we describe the main details of the proposed model. We also provide two examples of such a platform model: Table 4 and Table 5, show the platform models for the reference NEURAghe platform (see Section V-A) and the Jetson TX2 platform (see Section V-B), used in this work. The same description scheme may also apply to micro-controller based platforms, as long as the needed information regarding the working frequency and achievable performance, the local storage size and the communication resources are made available by the vendor in the related documentation. Our platform model is composed of three main elements:

- **Memory resources** (Row 2): specifies available on- and off-chip memory blocks, assigns an ID to each block, and indicates the related capacity;
- **IO channels** (Row 3): specifies connections that can be used to load/store input/output data to/from internal memories from/to external storage, assigns an ID to each connection, and lists the corresponding available bandwidth;
- **Processors** (Rows 4 to 16): a set of (parallel) processors, that represent the distributed heterogeneous computational resources, available on the platform, and can share the CNN workload. For the sake of brevity, in Table 4 and Table 5, we only show full processor description for the platform accelerators, and omit the full description of other processors, such as CPUs.

Every processor in our proposed platform model is composed of two main elements: a *processor description*, which describes the features of the processor, and a *computational model*, which indicates how the computational workload associated with CNN operators is deployed on the resources

TABLE 4. ALOHA platform model for NEURAghe.

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	
...			

specified in the platform model. The *processor description* (Rows 4 to 8 and Rows 15 to 16) describes the following parameters:

- **General characteristics** (Row 5 and Row 16): this section describes general characteristics of the processor, such as unique processor identifier (id), core type, and sub-type, top performance, and frequency.
- **Parallelism** (Row 6): this section describes how many operations in parallel can be executed by each core. To be compliant with what commonly happens in accelerators, where processing capabilities can often be implemented using multi-dimensional connected structures of processing elements, hierarchically organized, the available parallelism is described as an n-dimensional grid. The user must list n parallel factors corresponding to the hierarchy levels exposed by the processing element structures in the platform.
- **Power** (Row 7): this section contains optional information about the power consumption of the processor described. It may report a power consumption value in the active and idle state of the processor, and an energy

TABLE 5. ALOHA platform model for Jetson TX2.

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

cost per bit accessed in the global memory. This field is not needed if only latency is evaluated.

- **Overhead**: (Row 8) optional field allowing to account for programming cycles, required to start computations on the given processor.

The *computational model* (Rows 9 to 14) depicts the following parameters:

- **Loop nesting order and usage** (Row 10): specifies the order, in which loop of the layer computational tensor, given in Listing 1, explained in Section III-A, are executed on a specified processor.
- **Data transfers positioning** (Row 11): specifies the exact positions of data transfers in the layer computational tensor. As discussed in [47], this parameter can significantly affect the layer latency and energy consumption.
- **IO channels assignment** (Row 12): specifies the assignment of the platform data transfer channels to input data, output data, and weights, transferred during the layer execution.
- **Memory assignment** (Row 13): specifies the assignment of the platform memories to input data, output data, and weights, stored on the platform during the layer execution, and poses memory constraints on the loops of the computational tensor. This parameter is specified by the assignment of the platform memories of limited sizes to the loops of the layer computational tensor. By introducing this parameter, we model the impact of limited memory resources on the layer execution. The impact of the memory hierarchy is opposed to the loop unrolling and leads certain loops of the layer computational tensor to be *tiled*: partitioned in chunks that can be handled with the data fitting those memories.
- **Parallelism levels assignment** (Row 14): based on the computing units available on the platform, it describes how different degrees of parallelism are used to partially unroll the convolution loops. To correctly model the execution, we associate a computational tensor loop level to each dimension of the parallel computational grid of the platform accelerator.

A. EXAMPLE 1: DESCRIBING NEURAghe

NEURAghe is a CNN inference accelerator that can be configured at design time with different parameterization, but we consider in this paper a setup that is implemented 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 defined storage spaces, defined in the *Memory Resources* slot of Table 4. 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 among the hardware accelerator and the general-purpose processor. The data transfers between the global and local memories in the NEURAghe platform are handled through three separate DMA channels, transferring 8 B/cycle, described in Table 4 as *IO channel 0* and *1*, operating at 90 MHz, and *IO channel 2* operating at 180 MHz.

The computational resources of the NEURAghe platform consist of an ARM Cortex-a53 core exploited as a general-purpose processor and a convolution-specific FPGA-based accelerator. Due to the limited space, only the platform accelerator is fully described in Table 4.

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 models its computing resources by defining, in the *Parallelism* field, a *level*0 and *level*1 parallelism, respectively set to 9 and 10 and representing the dimensions of the computational grid, and a *level*2 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 [48], 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 [49], 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 dataflow 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 *level*0 parallelism is exploited to unroll computations over IFs, while the *level*1 parallelism, defines unrolling over OFs and *level*2 parallelism allows to 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.

B. EXAMPLE 2: DESCRIBING JETSON

Jetson TX2 [28] is a GPU-based platform from NVIDIA. The memory system in Jetson includes a unified 1.866-GHz DRAM memory, directly accessed and shared among 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, specified in Table 5 as *memory 0*, *memory 1*, and *memory 2*, respectively. Transfers between the platform global memory and other platform memories are handled through separate data transfer channels, described in Table 5 as *IO channels 0, 1, 2*, respectively. The computational resources of the Jetson TX2 platform are composed of an NVIDIA Pascal GPU, a quad-core Dual-Core NVIDIA Denver 2 64-Bit CPU, and a quad-Core ARM Cortex-A57 MPCore. Due to the limited space, only the platform GPU is fully described in Table 5. The GPU processor of the NVIDIA Jetson TX2 platform has two Streaming Multiprocessors (SMs). Each SM has 128 1.3-GHz cores and is capable of running 2048 threads, organized in 2048/128=16 thread blocks. The parallelism within the platform GPU is specified in the field *Parallelism* in Table 5, as parallelism levels 0, 1, and 2 of size 2 (SMs), 16 (blocks per SM) and 128 (threads per block), respectively. The peak performance of the NVIDIA Jetson TX2 GPU reaches 666.6 GOPs/s for FP32 CNN data precision (see field *General characteristics* in Table 5). Execution of a CNN on the Jetson TX2 platform is typically performed using the TensorRT DL framework [50], provided by NVIDIAs as an official DL framework for the platform. The TensorRT framework exploits the parallelism within the CNN layers, as specified in *Loop nesting order and usage* field in Table 5. As specified in field *Memory assignment*, during the CNN execution, the framework uses the global platform memory to store the output data and weights of CNN layers, and the shared GPU memory to store input data of CNN layers. When executed on the Jetson TX2 platform, computations within the CNN layers are limited by the sizes of the platform memories, as specified in field *Memory assignment* in Table 5.

VI. ALOHA EVALUATION PROCEDURE

In this section, we provide details about the ALOHA evaluation procedure. Our method ensures fast, yet accurate, evaluation of the CNN layers performance. Unlike other analytical methods with a similar level of abstraction, such as the Roofline model or OPS-based evaluation, discussed in Section II, our evaluation procedure captures the following important platform-aware factors, that affect platform-dependent metrics of every CNN layer executed on heterogeneous hardware platforms:

- **repeated transfers** of the layer input data and weights from the platform global memory to the local memories of the platform processors occur when the local platform memory, allocated to store the layer data and weights. cannot accommodate all the data and weights at once. Memory size also affects the amount of output data transferred from the local memory to the global one. The repetitive transfers cause additional time and energy overheads during the CNN layer execution;
- **occupancy/rounding effect**, i.e., a waste of computational power, caused by inefficient exploitation of the parallelism available in the platform, by the parallel computations to be carried out within a CNN layer [47]. Such waste is typically measured using wasted computational cycles, or partial processor occupancy, resulting in reduced performance of the platform computational resources [47];
- **separate bandwidth ceilings** reflect communication overheads, caused by an uneven distribution of the CNN layer data (input data, output data, and weights) over platform memories and data communication channels. Typically, the memory bandwidth of the platform is described by the peak memory bandwidth, which accounts for a high utilization of all data communication channels, available on the platform. However, in practice, not all data communication channels, available on the platform, are (efficiently) utilized, which leads to additional time overheads during the CNN layer execution.

Considering the aforementioned factors allows our method to achieve higher accuracy in evaluating platform-aware metrics of a CNN layer, compared to other analytical methods with a similar level of abstraction. The evaluation procedure involves three main phases:

- *Computational tensor generation*. This phase generates a representation of a CNN layer, annotated with an operator, input, and output data formats and weights, as a computational tensor, explained in Section III-A. It enables for explicit specification of the parallelism available within the CNN layer. The description of its functioning is placed in Section VI-A.
- *Computational tensor refinement*. In this phase, the generated platform-agnostic layer computational tensor

is enriched with platform-aware parameters of the ALOHA platform model, explained in Section V, and transformed into a platform-aware computational tensor. A detailed description of the steps performed in this phase, and of their effects, is given in Section VI-B.
- *Computational tensor analysis*. This phase involves analysis of the platform-aware computational tensor and final estimation of the platform-dependent metric of interest. Further description is given in Section VI-C.

To illustrate the phases of our evaluation procedure, we use as an example the convolutional layer l_1, shown in Figure 1, and explained in Section III-A, executed on the NEURAghe platform, represented as the ALOHA platform description in Table 4.

A. COMPUTATIONAL TENSOR GENERATION

In this phase, the ALOHA evaluation procedure represents the CNN layer as a 6-dimensional computational tensor, explained in Section III-A. To generate the layer computational tensor, the ALOHA method uses the generic CNN layer representation, given in Listing 1 and Table 3. For example, for the CNN layer l_1, shown in Figure 1, and explained in Section III-A, the ALOHA procedure generates the CNN layer computational tensor, provided in Listing 2.

B. COMPUTATIONAL TENSOR REFINEMENT

In this phase, the ALOHA evaluation procedure enriches the computational tensor of the CNN layer with platform-aware parameters of the ALOHA platform model, explained in Section V. The computational tensor refinement is performed in four steps (see Step 1 to Step 4 below). Steps 1 to 4 subsequently apply specific transformations to the computational tensor of the CNN layer. In this section, we show an example where the platform-agnostic computational tensor, given in Listing 2, explained in Section III-A, is refined with platform-aware details of the NEURAghe platform, represented using the ALOHA platform description in Table 4.

- **Step 1**: Apply **loop nesting order and usage** to the order of computational tensor loops. This step makes lines 1 and 2 in Listing 2 to swap places, resulting in Listing 3;
- **Step 2**: Apply **parallelism level assignment** to every computational tensor loop, unrolled over a dimension of a parallel computational grid. During this step, an indented loop *par*_∗, representing parallel computations, is inserted in the nested structure, according to the *Parallelism level assignment* field. The consequence of this action is that the number of iterations of the new couple of loops is rounded over the corresponding computational grid dimension. For example, the level 0 parallelism of size 9, shown in Table 4, and assigned to the IF loop of the computational tensor, causes the insertion of loop *par*_0 with 9 iterations in Listing 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, and assigned to the OF loop of the computational tensor, causes the insertion of loop *par*_1 with 10 iterations in Listing 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 the transfer of the layer input data, output data, and weights in the layer computational tensor. Every data transfer is assigned to a specific loop in the computational tensor, 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 whether the data is transferred from the main memory to the local processor memory ($action = load$), or from the local processor memory to the main platform memory ($action = store$). 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. The *data_bytes* parameter is assessed for every op/data type, using specific properties of the CNN layer and the layer computational tensor; mem_i specifies the platform memory, where data is accumulated; ch_j specifies the data transfer channel, used for transfer of the data. For example, in Listing 5, this step leads to the insertion of line 2, where input data of the CNN layer of size $9 * 28 * 7 * 4 * 2$ bytes is loaded from the device main memory into the processor local memory mem_0 through data communication channel ch_0. How to evaluate data transfer size is further detailed in Equation 7, introduced in the following phase, describing the *Computational tensor analysis*.
- **Step 4**: Pose **memory constraints** onto computational tensor loops. During this step, the evaluation procedure checks every loop, associated with a limited platform memory, as specified in the *Memory assignment* field of the platform model. If the utilization of the platform memory within the loop violates the memory constraint, the loop is tiled to avoid the violation. For example, as specified in Table 4, the OF loop of the computational tensor is limited by the local memory mem_1 of size 163840 bytes. In Listing 5, the layer tries to accumulate 815360 bytes in memory mem_1 (line 13), where $815360 > 163840$, and thus, Listing 5 violates a constraint, placed by memory mem_1 on the OF loop. This causes introduction of additional loop *out_tile* (line 1 in Listing 6) with 6 iterations, and reduction of the OF loop (line 6 in Listing 6) to $52/6 = 9$ iterations. In Listing 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*, and thus, does not violate the memory constraint, placed by memory mem_1 on the OF loop.

```
for i_feat in range(128):
 for o_feat in range(512):
  for fh in range(28):
   for fw in range(28):
    for k_y in range(1):
     for k_x in range(1):
      do MAC
```

Listing 3. Step 1. Loops nesting order and usage.

```
for i_feat in range(15):
 for par0 in range(9):
  for o_feat in range(52):
   for par1 in range(10):
    for fh in range(28):
     for fw in range(7):
      for par2 in range(4):
       for k_y in range(1):
        for k_x in range(1):
         do MAC
```

Listing 4. Step 2. Parallelism levels assignment.

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

Listing 5. Step 3. Data transfers introduction.

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

Listing 6. Step 4. Limits posing (tiling).

C. COMPUTATIONAL TENSOR ANALYSIS

In this phase, the ALOHA evaluation procedure analyses the platform-aware computational tensor, to estimate the total number of operations and data transfers, performed during the execution of a CNN layer. The total number of operations is computed using Equation 2 in Section III-B. According to Equation 2, the total number of operations OPS_{re}, performed by the refined CNN layer computational tensor, shown in Listing 6, is computed as: OPS_{re}[3] $= 6 * 15 * 9 * 9 * 10 * 28 * 7 * 4 * 1 * 1 * 2 \approx 110,07 * 10^6$.

[3]Description in Listing 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 III-A for the platform-agnostic CNN layer computational tensor, shown in Listing 2. The difference between OPS_{re} and OPS_{th} estimations occurs, because of an imperfect distribution of the layer computations over the platform processors. As a result, the CNN layer, distributed over the parallel computational resources of the platform (Listing 6), wastes some of the computational cycles, i.e., suffers 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, considering the platform-aware computational tensor allows assessing the actual amount of memory transfers, impacting the layer's operational intensity. Instead of using Equation 5, which 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, the 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:

$$Size_{re}(T) = \prod_{n=1}^{N} T.dim'_n * sizeof(pixel) * iterations_{tl} \quad (7)$$

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 convolution loops internal to the transfer level tl, which is subject to a certain number of $iterations_{tl}$, based on loop nesting structure.

Given the IO channel assignment, the ALOHA model evaluates the operational intensity over single available channels, exploiting Equation 3, where the OPS count is evaluated in details, considering rounding effects and tiling according to Listing 6, and the traffic value accounts for repeated transfers, exploiting Equation 7.

At this point, it is possible to use an approach inspired by the Roofline model, but turning Equation 6 into:

$$AP_{ALOHA} = min(AP_{max}, Int_{ch0} * bw_{ch0}, \ldots, Int_{chn} * bw_{chn}) \quad (8)$$

Considering $AP = AP_{ALOHA}$ in Equation 1, the execution time is evaluated as:

$$t_{ALOHA} = OPS_{re}/(AP_{ALOHA}) + ov \quad (9)$$

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

VII. CNN METRIC AGGREGATION

In this section, we present our CNN metric aggregation module. As explained in Section VI, the CNN metric aggregation module accepts as inputs estimations of the platform-dependent metrics of CNN layers and aggregates

them to deliver the final estimation of a platform-dependent metric of a CNN, such as CNN latency, CNN energy, and CNN throughput.

Despite systems with multiple accelerators being hardly available in the embedded domain, some research efforts have demonstrated that such a design technique can be useful [51], [52]. Thus our aggregation methodology enables the estimation to take into account an arbitrary number of processing elements. We do not account for contention on the off-chip memory, since previous experiments have shown that the effect of this issue is limited. References [51] shows this for the case of multiple NEURAghe instances insisting on the same DDR memory. We also assume, as in most of the approaches in the literature, communication with the host to be asynchronous, thus we do not consider the host CPU intervention to become a bottleneck.

Furthermore, in embedded systems, sensors are usually monitored with a specific frequency that is known at design time. Thus we assume input samples to be processed as soon as possible, as allowed by the throughput estimated by the method. In the case of variable sampling frequency, we assume design constraints to be defined according to the maximum rate of input samples and lower instant rates to be exploited for energy reduction using clock frequency scaling.

Along with the CNN, the platform specification, and the per-layer estimations of platform-dependent metrics, discussed in Section IV, the aggregation module accepts an optional *CNN execution configuration* input. This input consists of two optional parameters, characterizing the execution of a CNN on a target platform:

1) distribution of CNN operators over the target platform processors;
2) exploitation of task-level (pipeline) parallelism, available among layers of a CNN.

The distribution of CNN operators specifies how the CNN layers should be distributed over heterogeneous processors in a target platform, prospectively dedicated to different sets of operators (e.g. on an FPGA-based platform, where only some of the CNN operators can be executed on the FPGA, and the rest of the CNN operators are executed on the platform CPUs). Formally, we define this parameter 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 during the CNN execution, computations within every CNN layer l with $l.op = conv$ are performed only on the platform accelerator, and the computations within every CNN layer l with $l.op = gemm$ are performed only on the platform CPUs. If for a CNN operator op, no processor types are specified in the distribution op_dist, the aggregation module assumes, that every layer, performing operator op, can be executed on every processor, available on the target platform.

The exploitation of task-level (pipeline) parallelism specifies if a CNN is executed sequentially or as a pipeline. When a CNN is executed sequentially, only one of the CNN layers is executed at every moment in time. This type of CNN execution is typical for the majority of widely used DL frameworks such as PyTorch [37] or TensorFlow [38]. The execution of a CNN as a pipeline is an alternative to the sequential CNN execution. When a CNN is executed as a pipeline, several CNN layers can be executed in parallel, processing different inputs of a CNN. Execution of a CNN as a pipeline enables for higher throughput of CNNs, executed on heterogeneous target platforms [39], [40], and thus, it should be taken into account. Formally, we define the exploitation of task-level (pipeline) parallelism as a parameter $pipeline \in \{true, false\}$ with default value $pipeline = false$. If $pipeline = true$, a CNN is executed as a pipeline, otherwise the CNN is executed sequentially.

The CNN execution configuration is accepted as input by the *CNN schedule generator* module of the CNN metric aggregation module. The CNN schedule generator generates a schedule J for the input CNN. Schedule J assigns each layer $l_i \in L$ of the CNN a starting time $s_i \geq 0$ and a processor $PE_j, j \in [1, M]$ to be executed on. Currently, our CNN schedule generator can generate two types of schedule: 1) a sequential schedule, typical for CNNs, executed by widely used DL frameworks. This type of schedule is generated for CNN execution configurations with $pipeline = false$; 2) a pipeline schedule, proposed in [39], where a CNN is partitioned into M partitions, mapped onto M processors of the target platform, and all M CNN partitions are executed in parallel. This type of schedule is generated for the CNN execution configurations with $pipeline = true$. Additionally, if needed, one can extend our proposed CNN schedule generator or replace it with an alternative.

To generate a sequential schedule, our aggregation module uses Algorithm 1. Algorithm 1 accepts as inputs: 1) a CNN; 2) set of processors $PE = \{PE_1, PE_2, \ldots PE_M\}$, available on the target platform; 3) a distribution of the CNN operators over the target platform processors op_dist; 4) a set of per-layer latency estimations $\{t(l_i, PE_j)\}, i \in [1, L], j \in [1, M]$. As output, Algorithm 1 provides a CNN schedule, represented as a set of pairs (s_i, PE_j), where s_i is the starting time of a layer $l_i \in L$; $PE_j \in PE$ is a processor of the target platform. In Line 1, Algorithm 1 defines an empty schedule J and sets current starting time s to 0. In Lines 2 to 17, Algorithm 1 assigns time $s_i \geq 0$ and processor $PE_j \in PE$ to every layer $l_i \in L$ of a CNN. In Lines 3 to 9, Algorithm 1 defines list of processors $PE_{suitable}$, suitable for execution of layer l_i. If operator op, performed by layer l_i is specified in the distribution op_dist, list of suitable processors $PE_{suitable}$ is defined in Lines 5 to 7 as a list of all processors, that support operator op. Otherwise, $PE_{suitable}$ is defined in Line 9 as a list of all processors, available on the platform. In Lines 10 to 14, Algorithm 1 selects processor PE_j from the list of suitable processors $PE_{suitable}$, such that execution of layer l_i on the processor PE_j leads to the smallest latency $t(l_i, PE_j)$ of layer l_i. In Lines 15 to 17, Algorithm 1 assigns time $s_i = s$ and processor PE_j to the layer l_i (Line 15) and increases starting time s by the latency $t(l_i, l_j)$ of layer l_i,

Algorithm 1: Sequential Schedule Generation

Input: $CNN(L, E), PE, op_dist, \{t(l_i, PE_j)\}$
Result: CNN schedule J

1 $J = \emptyset; s = 0;$
2 **for** $i \in [1, |L|]$ **do**
3 $\quad PE_{suitable} = \emptyset;$
4 $\quad$ **if** $\exists(op, proc_type) \in op_dist : op = l_i.op$ **then**
5 $\quad\quad$ **for** $(l_i.op, proc_type) \in op_dist$ **do**
6 $\quad\quad\quad$ **for** $PE_j \in PE : PE_j.type = proc_type$ **do**
7 $\quad\quad\quad\quad PE_{suitable} = PE_{suitable} + PE_j;$
8 $\quad$ **else**
9 $\quad\quad PE_{suitable} = PE;$
10 $\quad PE_j = PE_{suitable}.pop();$
11 $\quad$ **while** $PE_{suitable} \neq \emptyset$ **do**
12 $\quad\quad PE_k = PE_{suitable}.pop();$
13 $\quad\quad$ **if** $t(l_i, PE_k) < t(l_i, PE_j)$ **then**
14 $\quad\quad\quad PE_j = PE_k;$
15 $\quad s_i = s;$
16 $\quad J = J + (s_i, PE_j);$
17 $\quad s = s + t(l_i, PE_j);$
18 **return** J

executed on processor PE_j. Finally, in Line 18, Algorithm 1 returns sequential schedule of the input CNN.

To generate a pipeline schedule, our aggregation module uses the heuristic algorithm, proposed in [39]. As explained above, the pipeline schedule can affect the CNN throughput. Our proposed CNN metric aggregation module captures the impact of the CNN schedule on the CNN throughput by considering the CNN schedule during the CNN throughput estimation (see Equation 11 explained below).

The CNN schedule, generated by the CNN schedule generator, is accepted as input by the CNN metric aggregation sub-module, along with per-layer CNN metric evaluations. The CNN metric aggregation sub-module, uses Equation 10, Equation 11 and Equation 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.

$$t_{CNN} = s_{|L|} + t(l_{|L|}) - s_1 \tag{10}$$

$$Th_{CNN} = \begin{cases} 1/max_j \sum_{(s_i, PE_j) \in J} t(l_i) & ifpipeline \\ 1/t_{CNN} & otherwise \end{cases} \tag{11}$$

$$En_{CNN} = \sum_{(s_i, PE_j) \in J} t * P_{act} + t_{idle} * P_{idle} + b_{acc} * En_{bit} \tag{12}$$

In Equation 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 CNN layer l_1; Latency $t(l_i)$ of CNN layer l_i is estimated by the ALOHA per-layer evaluation procedure, proposed in Section VI.

In Equation 11, the CNN throughput is computed. If a CNN is executed as a pipeline, its throughput is estimated as 1, divided on time $max_j \sum_{(s_i, PE_j) \in S} t(l_i)$, required to execute CNN on processors $\{PE_j\}, j \in [1, M]$ of the target platform, where $\sum_{(s_i, PE_j) \in S} t(l_i)$ is time (in seconds), taken by processor PE_j to execute all CNN layers $\{l_i\}$, mapped on this processor. If a CNN is executed sequentially, its throughput is computed as $1/t_{CNN}$, where t_{CNN} is the CNN latency, computed using Equation 10.

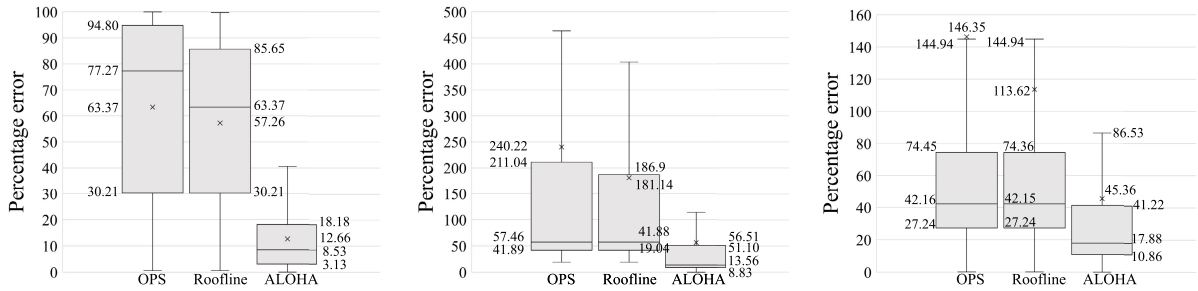
In Equation 12, the total CNN energy En_{CNN} is computed as the sum of energy costs of all layers of a CNN. The energy cost of layer $l_i \in L$ is computed as the sum of three factors: the layer latency t, which is actually a function $t(l_i, PE_j)$ of the layer l_i and of the processor PE_j on which it is executed, multiplied on peak power consumption P_{act}, which is depending on the processor PE_j; the idle time t_{idle}, which is defined according to a latency constraint and the layer latency t, multiplied on idle power consumption P_{idle}; the cost En_{bit} of bit accesses to the global memory, multiplied on the number of bits b_{acc} transferred by the processor PE_j during the execution of layer l_i.

VIII. EXPERIMENTAL RESULTS

In the following, we present experimental results involving execution time and energy consumption predictions obtained by the ALOHA method. In section VIII-A, the accuracy of our proposed method is compared with the OPS count and the Roofline model in a single layer execution time estimation, showing reduced average prediction error in both evaluated platforms, NEURAghe and Jetson. In section VIII-B, we consider a consumption model characterized for NEURAghe, and evaluate the advantages of the ALOHA method, over the Roofline model, in providing accurate execution time and memory access count predictions for the energy consumption estimation. In section VIII-C, we consider a NAS process, aiming at selecting optimal CNN architectures for both target platforms, NEURAghe and Jetson. The last section VIII-D explores throughput estimations for CNNs executed on a heterogeneous platform, such as Jetson TX2. We evaluate the combined impact of layer-level ALOHA prediction accuracy and the proposed CNN metric aggregation when different scheduling schemes are exploited. 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.

A. LAYER-LEVEL ACCURACY

We characterized a grid of over 2000 common CNN convolutional layer configurations, whose parameters are summarized in Table 6, to quantitatively compare the presented ALOHA method with the OPS count, and the traditional Roofline model, in execution time estimation. Figure 3a and 3b show the prediction error distribution for the three methods, through comparison with execution time measured on the target platforms.



(a) NEURAghe (b) Jetson (c) Batch processing on Jetson

FIGURE 3. Error distribution on the latency estimation for the examined estimation methods.

TABLE 6. 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.

	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

NEURAghe. The rounding effects on the layer's parameters, connected to the computing matrix size, deeply affect execution time prediction. Neglecting such an effect leads to dramatically underestimate the actual number of OPS performed during the layer execution, by a factor of 0.25 on average, and up to a factor of over 0.85. To highlight the contribution of the other non-idealities modeled by our approach, the rounding effect correction was also considered in the Roofline and the OPS-based estimations. Nonetheless, as shown in Figure 3a, the OPS count method provides latency estimations suffering from 63.4% average error, despite being very immediate and comfortable to build. The Roofline model, although introducing rough data transfer time evaluation considering the IO bandwidth ceiling, still shows 57.3% estimation error. On the other hand, NEURAghe's ALOHA model proves to be significantly more accurate, reducing the average estimation error to 12.7%.

Jetson. The runtime management in the GPU engine is intrinsically less predictable than the hardware-based scheduling in NEURAghe. The ALOHA method only provides the possibility to account for inefficiencies connected to the Operating System in terms of startup time, modeled for Jetson as a constant startup overhead. Thus, the unpredictability of the runtime management results in a less accurate platform model, compared to the one obtained for NEURAghe. The evaluated layers in Table 6 were implemented with the TensorRT [50] Deep Learning library, which is the best-known and state-of-the-art Deep learning library for the NVIDIA Jetson TX2 platform. The estimation error affecting the examined methods, depicted in Figure 3b, shows how both the OPS count and the Roofline model provide very poor precision, with up to 2x times over-estimation of the CNN layers performance, compared to the performance measured on the platform, while the ALOHA model shows reduced 56.5% average error.

Impact of batch processing on Jetson. To account for other processing scenarios, we consider the case of batch processing on the Jetson platform. Figure 3c reports the estimation error of the examined methods, considering variable batch sizes, up to a value of 32. This scheduling choice allows for better resource utilization, providing greater opportunities for parallelization. This results in a measured operating performance 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 2x the one obtainable thanks to the ALOHA method.

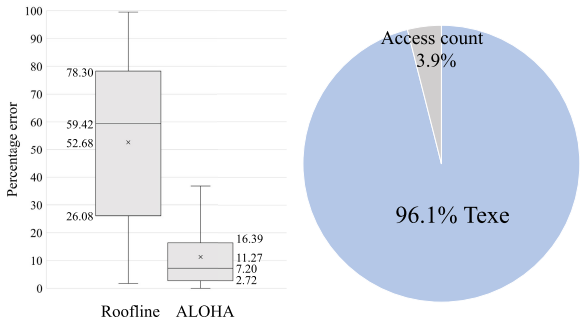
B. IMPACT ON ENERGY CONSUMPTION ESTIMATION

The scope of this section is to provide an overview of the impact that the ALOHA method has on energy consumption estimation. To this aim, we have considered an energy consumption expression that highlights the dependence of this metric on execution time and memory access count, as represented in Equation 13, which is a simplified version of 12 considering only the power dissipated by a convolutional layer executed on the accelerator:

$$En = P_{act} * t + En_{bit} * b_{acc} \tag{13}$$

For single layer estimation we neglect the idle contribution, considering $t_{idle} = 0$ in Equation 12. As an example of the impact of detailed platform modelling, we have characterized Equation 13 for NEURAghe, using the values reported in Table 4 for P_{act} and En_{bit}. For brevity reasons, we only refer to the NEURAghe platform, although 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.

Exploiting the model in Equation 13, we have estimated the energy consumption for the grid of layers with parameters summarized in Table 6, by referring to access count and execution time predictions based on the Roofline and ALOHA methods. The prediction error is evaluated through



(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. Error on NEURAghe energy consumption estimation.

comparison with estimates relying on measured execution time and precise access count. The results in terms of error distribution are summarized in Figure 4a. The ALOHA method provides more accurate execution time predictions, and a precise evaluation of data transfers, producing an average 11.3% estimation error on the energy consumption value. The Roofline model estimations are affected by an average error of around 52.7%, which is mainly connected to the predicted execution time, as shown in Figure 4b. Neglecting repeated transfers, in evaluating the number of accesses to the DDR, produces an average 17% error on the memory traffic evaluation, although it only determines 2% prediction error on the energy estimation.

C. IMPACT ON NAS

We analyze here the impact of platform awareness and accurate inference execution time prediction in a NAS process, aiming at selecting optimal CNN architectures for the target platforms modeled in the previous sections, NEURAghe and Jetson.

The selected search space explores network architectures for image classification, exploiting the structure of the well known VGG architecture [53], targeting CIFAR-10 [29]. The considered network architectures are composed as indicated in Table 7. 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 7.

The search strategy first exploits one-shot training as developed by the authors of [34], to train all the possible combinations of the parameters in Table 7. The total number of trained networks sums up to 3.16M design points.

Once the training is completed, an evolutionary algorithm is exploited to search for the optimal network architecture.

TABLE 7. Design Space Exploration parameters for NAS targeting NEURAghe and Jetson.

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

The training procedure was executed on Azure NC6_v2 Virtual Machine, exploiting an NVIDIA Tesla P100 GPU. According to the Progressive Shrinking method exploited by [34], the teacher network was trained for 15 epochs, then subnetworks with different depth values were refined through additional 45 epochs, while different values of channel width, as described in Table 7, are lastly enabled through the final 180 epochs. The algorithm starts from the random selection of 100 network architectures, which constitute the starting population. At each evolution step, the architectures in the population are evaluated and a new generation is created depending on the evaluation results, as the union of three components:

- 25 most accurate architectures of the previous generation;
- 50 architectures obtained through their random mutation in stage depth and layer width;
- 25 architectures obtained through the crossover of the top 2 most accurate architectures of the previous generation.

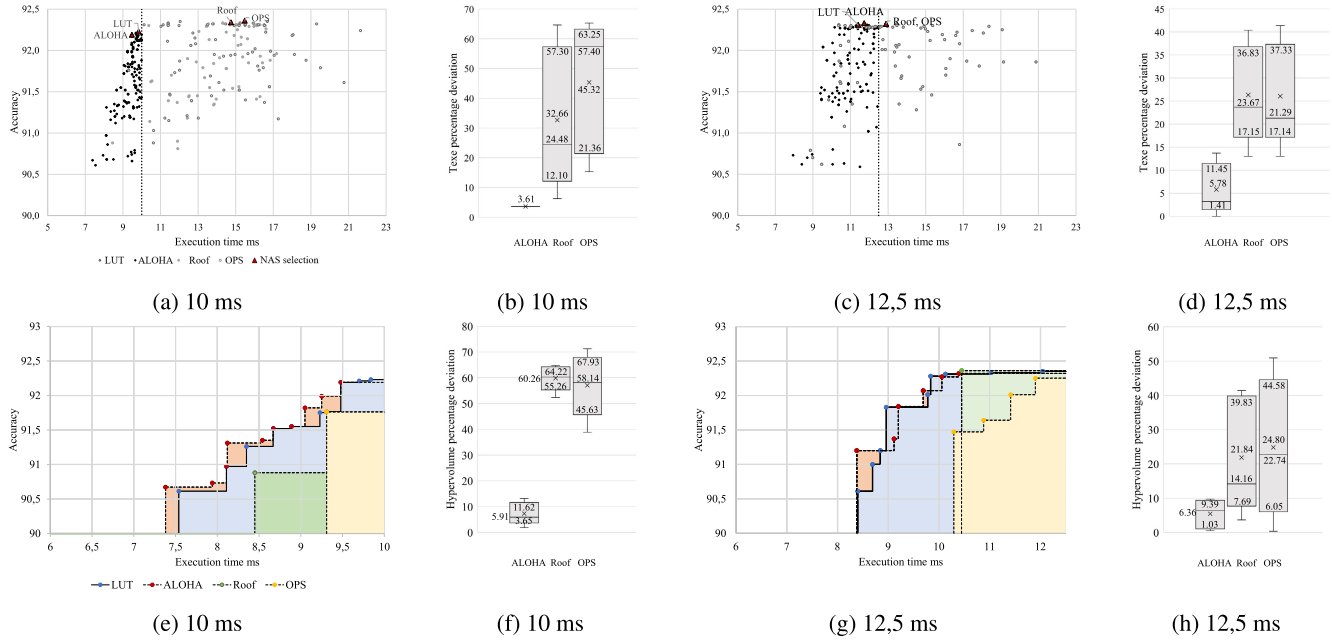
According to the implementation in [34], the latency constraint limits a platform-aware search space [54]. Network architectures are only admitted in evolving generations if they are compliant with the latency constraint.

The results presented in the following involve selections targeting best network accuracy, where applying the latency constraint at each generation exploits OPS, Roofline, or ALOHA methods. We compare the resulting architecture selections, with those obtained using LUTs for the latency evaluation. Since LUTs contain latency numbers measured on the actual hardware, this method, although being not flexible, is very precise and it is considered as a reference in the following. The LUTs were populated by performing latency on-hardware measurements on both NEURAghe and Jetson platforms. 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 an independent selection output, after 20 generations.

Every NAS experiment is repeated 5 times, to account for the effect of random selections within the genetic algorithm.

NAS targeting NEURAghe. We have performed two NAS processes, using respectively 10 ms and 12.5 ms as latency constraints.

For each constraint, we have executed the whole selection process using each of the four latency estimation methods presented above and we compared the selection results.



(a) 10 ms (b) 10 ms (c) 12,5 ms (d) 12,5 ms

(e) 10 ms (f) 10 ms (g) 12,5 ms (h) 12,5 ms

FIGURE 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. e-g) Hypervolume comparison for the Pareto fronts resulting from evolutionary NAS. Performance evaluation is based on LUTs, and on the evaluated prediction methods. f-h) Hypervolume deviation of the Pareto fronts resulting from NAS based on the examined prediction methods, from the Pareto front produced in NAS exploiting LUTs.

The Pareto plots in Figure 5a and 5c show the distribution of the design points in the last generation, in one of the five trials: the design points selected by the NAS based on the ALOHA method are much closer to the points selected by the NAS according to LUTs; on the contrary, both the OPS and Roofline methods mistakenly focus on complex architectures, with higher levels of accuracy, but violating the search constraint on execution time. However, while the ALOHA pattern in Figure 5a looks very different from the one of the evaluated alternatives, the example in Figure 5c shows that the accuracy in inference time prediction has a lower impact on the design points selection when the latency constraint is more relaxed.

Figure 5b and 5d, show how different the CNN architectures selected using the evaluation methods are from the CNN selected by LUT, over the five trials. In general, in terms of latency, the solution found using ALOHA is significantly more similar (around 3% deviation on average for the 10 ms constraints and always few percentage points for 12 ms).

To quantitative estimate the similarity of the explorations, besides the final selection points, we have built the Pareto front resulting from each of the NAS processes and referred to common metrics as the Degree of Approximation [55] and the Hypervolume [56] to compare them. The DoA values, reported in Table 8, show that 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 5e and 5g show the hypervolume shapes, in the admissible region, of the fronts obtained in one of NAS trials, for each of the constraints defined. The hypervolumes in this section are evaluated by choosing a reference point aligned with the constraint, with coordinates (90%, *constraint* ms). The ALOHA prediction produces a pattern similar to LUTs, while the Roofline and OPS count methods result in significantly different hypervolume shapes. Figure 5f and 5h report the deviation of the hypervolume indicator from the one evaluated on the LUT Pareto front, throughout the set of trials, confirming that the ALOHA Pareto front shapes are, in all the trials, much more similar to LUT compared to the alternatives.

TABLE 8. 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 DoA	Roofline DoA	**ALOHA DoA**
10 ms	0.54	0.63	**0.02**
12.5 ms	0.51	0.47	**0.04**

Finally, we compare the methods also in terms of predictability and reliability, since, when using inaccurate estimation methods, during the selection process, the algorithm could include in evolving populations design points that are wrongly estimated to be compliant with the constraints. We have counted how often the architectures included in the last population are instead inadmissible according to their on-hardware measure. Table 9 shows ALOHA has selected only CNN architectures compliant with the latency constraint, at the end of all the five trials (100% of selected admissible points). On the contrary, OPS- and Roofline-based selections include a quite high rate of inadmissible points: only around 3% of the points are legal at 10 ms and only around 30% for 12.5 ms.

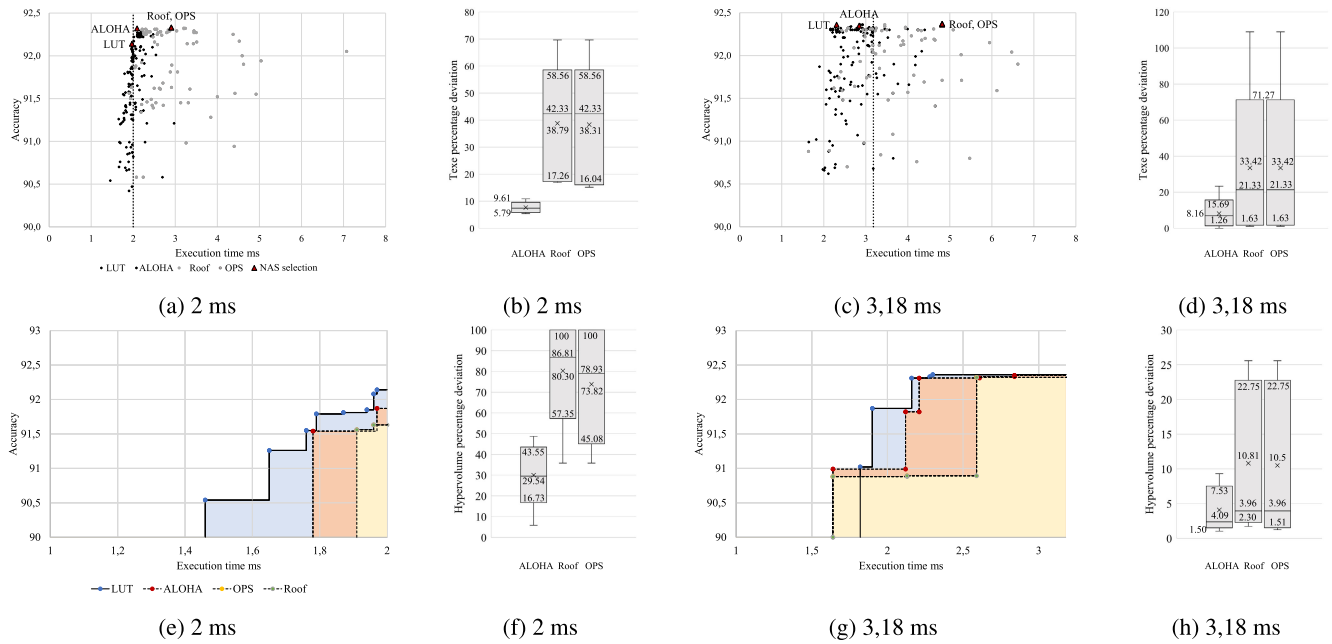


FIGURE 6. Pareto plot of accuracy vs latency for the design points in the last generation of one trial of evolutionary NAS targeting Jetson. Performance evaluation exploits LUTs or ALOHA, Roofline, and OPS based estimation. b-d)Latency deviation distribution, among 5 trials, of NAS selection. Performance evaluation based on the ALOHA, Roofline, and OPS models is compared to NAS selection obtained by performance evaluation based on LUTs. e-g)Hypervolume comparison for Pareto fronts resulting from evolutionary NAS. Performance evaluation is based on LUTs, and on the evaluated prediction methods. f-h)Hypervolume deviation of the Pareto fronts resulting from prediction methods, from the Pareto front produced in NAS exploiting LUTs.

TABLE 9. 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.

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

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. Figure 6a and Figure 6c highlight that, although in general the ALOHA method is less accurate on Jetson, the discrepancy between the selection operated using LUTs and ALOHA is still reduced compared to the alternatives. Figure 6b shows that this has been the case in all the trials when considering the tightest constraint, while Figure 6d shows that when the constraint is softened, the inaccuracy in performance estimation has a lower impact. Nevertheless, on average, the ALOHA method has produced results clearly more aligned with what was produced by LUTs. The comparison based on the Pareto fronts confirms the same trend. Hypervolumes in Figure 6e and 6g show that ALOHA finds Pareto-optimal points that better follow the LUT's Pareto front profile, especially in the case of the 2ms constraint. In this case, the ALOHA-driven Hypervolume indicator differs from the LUT-driven one, in general, by less than 10%, while, in their more favorable cases, the alternatives differ by at least 40%. The DoA metric reported in Table 10 provides similar results since ALOHA reduces deviation by at least a factor of 4. When the constraint is more relaxed the benefits are, as expectable, less visible. ALOHA

TABLE 10. 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.

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

TABLE 11. 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.

Constraint	OPS	Roofline	**ALOHA**
10 ms	2.6%	2.1%	**29.6%**
12.5 ms	61.9%	61.9%	**76%**

reduces DoA by 2× and Hypervolume deviation by around 3×, on average. Table 11 provides a view of the effects of ALOHA, when exploring for Jetson, on predictability and reliability. Considering the tightest constraint, when using Roofline and OPS, only around 3% of the architectures in the last population are effectively legal. ALOHA, in this case, is also selecting some inadmissible points, however, the rate of legal points at the end of the process is one order of magnitude higher. 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%).

D. IMPACT OF AGGREGATION ON PREDICTION ACCURACY

Finally, in this section, we evaluate the impact of CNN metric aggregation, proposed in Section VII, on the prediction

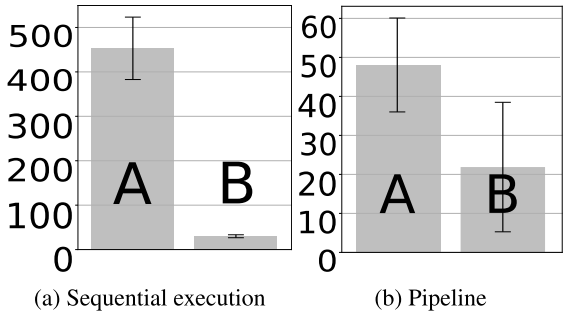


FIGURE 7. Error distribution on predicted throughput for CNNs distributed over heterogeneous processors of Jetson TX2.

accuracy of our proposed methodology. We perform an experiment, where we use our proposed evaluation method with different CNN execution configurations, to estimate the throughput of over 1700 common CNNs, resulting from NAS exploration described in Section VIII-C. The structure of these CNNs is summarized in Table 7. Experiments in this section focus on the aggregation of throughput of the CNNs, executed on Jetson TX2 heterogeneous embedded platform [28], with different ways of CNN execution, discussed in Section VII. In this experiment, we perform two trials.

In Trial 1, we study the impact of CNN distribution over platform processors, discussed in Section VII on evaluation of platform-aware metrics of CNNs. In this trial, we estimate throughput of the CNNs, when layers of every CNN are executed sequentially (one-by-one) and are distributed over a GPU and 4 ARM Cortex A-57 CPUs of the Jetson TX2 platform, so that the computations within every layer $l_i : op_i = conv$ are offloaded on the platform GPU, and computations within every layer $l_i : op_i \neq conv$ are 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 specified as $pipeline = false$ and $ops_dist = \{(conv, accelerator), (gemm : CPU), (pool : CPU)\}$. The results of this experiment are given in Figure 7a using mean-and-error. The CNN throughput estimation is very inaccurate when the layers distribution is not considered. Error, on average, reaches 450% when using estimation A. In B, considering the execution configuration, proposed in our CNN metric aggregation (see Section VII), our method takes into account heterogeneity and reduces error down to 27%.

In Trial 2, we study the impact of pipeline parallelism exploitation, discussed in Section VII on the evaluation of platform-aware metrics of CNNs. In this trial, we estimate throughput of the CNNs, when layers are distributed over all processors in the platform exploiting pipeline parallelism. We compare the error in CNN throughput estimation, with respect to the throughput measured on hardware, when the CNN execution configuration is specified as: A) $pipeline = false$ and $ops_dist = \emptyset$; B) $pipeline = true$ and $ops_dist = \emptyset$. The results of this experiment are given in Figure 7b. The CNN throughput estimation error, on average, reaches 48% when using estimation A, unaware of the parallel execution, and 21% when using estimation B, which considers the exploitation of pipeline.

IX. CONCLUSION

We proposed the ALOHA method, as a general and flexible instrument to provide accurate latency, energy, and throughput estimations of a given CNN architecture executed on a target hardware platform, by exploiting easy-to-use platform and computational models, introducing platform awareness without requiring access to on-hardware measurements. We showed it allows for a reduction of 3x, up to 5x, of the average layer-level latency estimation error affecting common alternative analytical methods and evaluated on two different platforms: an FPGA-based accelerator, NEURAghe, and a GPU-based platform, Jetson TX2. Moreover, the proposed method allows to model execution on heterogeneous platforms, considering different mappings and scheduling schemes of the CNN computations on the platform's processing resources, and providing accurate system-level throughput estimations. The accuracy in execution modeling and latency estimation was also evaluated in its impact on the energy consumption estimation, resulting in a 2x precision improvement. Finally, we show that the high level of platform awareness provided by detailed modeling through ALOHA improves by a factor of 4x NAS output predictability, when compared to the OPS count and Roofline models, and leads to select Pareto optimal points close to the ones evaluated in measurement-based NAS.

ACKNOWLEGMENT

(Paola Busia and Svetlana Minakova are co-first authors.)

REFERENCES

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, May 2015.

[2] M. Z. Alom, T. M. Taha, C. Yakopcic, S. Westberg, P. Sidike, M. S. Nasrin, B. C. Van Esesn, A. A. S. Awwal, and V. K. Asari, "The history began from AlexNet: A comprehensive survey on deep learning approaches," 2018, *arXiv:1803.01164*. [Online]. Available: https://arxiv.org/abs/1803.01164

[3] T.-D. Do, M.-T. Duong, Q.-V. Dang, and M.-H. Le, "Real-time self-driving car navigation using deep neural network," in *Proc. 4th Int. Conf. Green Technol. Sustain. Develop. (GTSD)*, Nov. 2018, pp. 7–12.

[4] B. Savelli, A. Bria, M. Molinara, C. Marrocco, and F. Tortorella, "A multi-context CNN ensemble for small lesion detection," *Artif. Intell. Med.*, vol. 103, Mar. 2020, Art. no. 101749.

[5] J. Ahn, J. Paek, and J. Ko, "Machine learning-based image classification for wireless camera sensor networks," in *Proc. IEEE 22nd Int. Conf. Embedded Real-Time Comput. Syst. Appl. (RTCSA)*, Aug. 2016, p. 103.

[6] S. Branco, A. G. Ferreira, and J. Cabral, "Machine learning in resource-scarce embedded systems, FPGAs, and end-devices: A survey," *Electronics*, vol. 8, no. 11, p. 1289, 2019.

[7] D. Liu, H. Kong, X. Luo, W. Liu, and R. Subramaniam, "Bringing AI to edge: From deep learning's perspective," 2020, *arXiv:2011.14808*. [Online]. Available: https://arxiv.org/abs/2011.14808

[8] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, Jun. 2017, pp. 1–12, doi: 10.1145/3079856.3080246.

[9] E. Azarkhish, D. Rossi, I. Loi, and L. Benini, "Neurostream: Scalable and energy efficient deep learning with smart memory cubes," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 2, pp. 420–434, Feb. 2018.

[10] G. Desoli, N. Chawla, T. Boesch, S. P. Singh, E. Guidetti, F. D. Ambroggi, T. Majo, P. Zambotti, M. Ayodhyawasi, H. Singh, and N. Aggarwal, "14.1 A 2.9TOPS/W deep convolutional neural network SoC in FD-SOI 28 nm for intelligent embedded systems," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, San Francisco, CA, USA, Feb. 2017, pp. 238–239.

[11] Movidius. (2020). *Movidius Neural Compute Stick: Accelerate Deep Learning Development at the Edge*. [Online]. Available: https://developer.movidius.com/

[12] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Seoul, South Korea, Jun. 2016, pp. 367–379.

[13] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *Proc. Int. Conf. Mach. Learn.*, 2019, pp. 6105–6114.

[14] C. Banbury, C. Zhou, I. Fedorov, R. Matas Navarro, U. Thakker, D. Gope, V. Janapa Reddi, M. Mattina, and P. N. Whatmough, "MicroNets: Neural network architectures for deploying TinyML applications on commodity microcontrollers," 2020, *arXiv:2010.11267*. [Online]. Available: http://arxiv.org/abs/2010.11267

[15] A. Gordon, E. Eban, O. Nachum, B. Chen, H. Wu, T. Yang, and E. Choi, "MorphNet: Fast & simple resource-constrained structure learning of deep networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 1586–1595.

[16] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for floating point programs and multicore architectures," *Commun. ACM*, vol. 52, no. 2, pp. 65–76, 2009, doi: 10.1145/1498765.1498785.

[17] H. Cai, L. Zhu, and S. Han, "ProxylessNAS: Direct neural architecture search on target task and hardware," in *Proc. Int. Conf. Learn. Represent.*, 2019. [Online]. Available: https://openreview.net/forum?id=HylVB3AqYm

[18] L. L. Zhang, Y. Yang, Y. Jiang, W. Zhu, and Y. Liu, "Fast hardware-aware neural architecture search," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. Workshops (CVPRW)*, Jun. 2020, pp. 2959–2967.

[19] W. Jiang, L. Yang, E. Sha, Q. Zhuge, S. Gu, Y. Shi, and J. Hu, "Hardware/software co-exploration of neural architectures," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 12, pp. 4805–4815, Dec. 2020.

[20] Q. Lu, W. Jiang, X. Xu, Y. Shi, and J. Hu, "On neural architecture search for resource-constrained hardware platforms," 2019, *arXiv:1911.00105*. [Online]. Available: http://arxiv.org/abs/1911.00105

[21] C. Hao, X. Zhang, Y. Li, S. Huang, J. Xiong, K. Rupnow, W.-M. Hwu, and D. Chen, "FPGA/DNN co-design: An efficient design methodology for IoT intelligence on the edge," in *Proc. 56th Annu. Design Automat. Conf.*, New York, NY, USA, 2019, pp. 1–6, doi: 10.1145/3316781.3317829.

[22] L. Yang, Z. Yan, M. Li, H. Kwon, W. Jiang, L. Lai, Y. Shi, T. Krishna, and V. Chandra, "Co-exploration of neural architectures and heterogeneous asic accelerator designs targeting multiple tasks," in *Proc. 57th ACM/IEEE Design Automat. Conf. (DAC)*, Jul. 2020, pp. 1–6.

[23] A. Marchisio, A. Massa, V. Mrazek, B. Bussolino, M. Martina, and M. Shafique, "NASCaps: A framework for neural architecture search to optimize the accuracy and hardware efficiency of convolutional capsule networks," in *Proc. 39th Int. Conf. Comput.-Aided Design*, New York, NY, USA, 2020, pp. 1–9, doi: 10.1145/3400302.3415731.

[24] H. Qi, E. R. Sparks, and A. Talwalkar, "Paleo: A performance model for deep neural networks," in *Proc. 5th Int. Conf. Learn. Represent. (ICLR)*, Toulon, France, 2017. [Online]. Available: https://openreview.net/forum?id=SyVVJ85lg

[25] D. Justus, J. Brennan, S. Bonner, and A. S. McGough, "Predicting the computational cost of deep learning models," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2018, pp. 3873–3882.

[26] C.-C. Wang, Y.-C. Liao, M.-C. Kao, W.-Y. Liang, and S.-H. Hung, "PerfNet: Platform-aware performance modeling for deep neural networks," in *Proc. Int. Conf. Res. Adapt. Convergent Syst.*, New York, NY, USA, 2020, pp. 90–95, doi: 10.1145/3400286.3418245.

[27] P. Meloni, A. Capotondi, G. Deriu, M. Brian, F. Conti, D. Rossi, L. Raffo, and L. Benini, "NEURAGHE: Exploiting CPU-FPGA synergies for efficient and flexible CNN inference acceleration on Zynq SoCs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 11, no. 3, p. 18, Dec. 2018, doi: 10.1145/3284357.

[28] N. Corporation. *Jetson TX2 Platform Technical Specification*. Accessed: Apr. 7, 2021. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-tx2/

[29] A. Krizhevsky. (2009). *Learning Multiple Layers of Features From Tiny Images*. [Online]. Available: https://www.cs.toronto.edu/kriz/learning-features–TR.pdf

[30] J. Czaja *et al.*, "Applying the Roofline model for deep learning performance optimizations," 2020, *arXiv:2009.11224*. [Online]. Available: https://arxiv.org/abs/2009.11224

[31] N. K. Jha and S. Mittal, "Modeling data reuse in deep neural networks by taking data-types into cognizance," *IEEE Trans. Comput.*, vol. 70, no. 9, pp. 1526–1538, Sep. 2021.

[32] X. Luo, D. Liu, H. Kong, and W. Liu, "EdgeNAS: Discovering efficient neural architectures for edge systems," in *Proc. IEEE 38th Int. Conf. Comput. Design (ICCD)*, Oct. 2020, pp. 288–295.

[33] C.-H. Hsu, S.-H. Chang, D.-C. Juan, J.-Y. Pan, Y. Chen, W. Wei, and S.-C. Chang, "MONAS: Multi-objective neural architecture search using reinforcement learning," 2018, *arXiv:1806.10332*. [Online]. Available: https://arxiv.org/abs/1806.10332

[34] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han, "Once-for-all: Train one network and specialize it for efficient deployment," in *Proc. 8th Int. Conf. Learn. Represent. (ICLR)*, Addis Ababa, Ethiopia, Apr. 2020. [Online]. Available: https://openreview.net/forum?id=HylxE1HKwS

[35] B. Wu, K. Keutzer, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, and Y. Jia, "FBNet: hardware-aware efficient ConvNet design via differentiable neural architecture search," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2019, pp. 10726–10734.

[36] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, "GPGPU performance and power estimation using machine learning," in *Proc. IEEE 21st Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2015, pp. 564–576.

[37] A. Paszke, *Pytorch: An Imperative Style, High-Performance Deep Learning Library*. Red Hook, NY, USA: Curran Associates, 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-pe rformance-deep-learning-library.pdf

[38] M. Abadi. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. [Online]. Available: http://tensorflow.org/

[39] S. Wang, A. Pathania, and T. Mitra, "Neural network inference on mobile SOCs," *IEEE Des. Test*, vol. 37, no. 5, pp. 50–57, Oct. 2020.

[40] 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*, Samos, Greece. Springer, 2020, pp. 18–35.

[41] C. Shea and T. Mohsenin, "Heterogeneous scheduling of deep neural networks for low-power real-time designs," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 15, no. 4, p. 36, 2019.

[42] L. Lai, N. Suda, and V. Chandra, "Not all ops are created equal!" *CoRR*, vol. abs/1801.04326, 2018. [Online]. Available: http://arxiv.org/abs/1801.04326

[43] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, "MnasNet: Platform-aware neural architecture search for mobile," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2019, pp. 2815–2823.

[44] C. Kyrkou, G. Plastiras, T. Theocharides, S. I. Venieris, and C.-S. Bouganis, "DroNet: Efficient convolutional neural network detector for real-time UAV applications," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 967–972.

[45] H. Kwon, P. Chatarasi, M. Pellauer, A. Parashar, V. Sarkar, and T. Krishna, "Understanding reuse, performance, and hardware cost of DNN dataflow: A data-centric approach," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, New York, NY, USA, 2019, pp. 754–768, doi: 10.1145/3352460.3358252.

[46] A. Tools Community. *Open Neural Network Exchange (ONNX)*. Accessed: Apr. 7, 2021. [Online]. Available: https://onnx.ai/

[47] N. Corporation. (2021). *CUDA C++ Best Practices Guide*. [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUDACBestPracticesGuide.pdf

[48] Xilinx. *Xilinx Power Estimator*. Accessed: Apr. 7, 2021. [Online]. Available: https://www.xilinx.com/products/technology/power/xpe.html

[49] K. Chandrasekar, C. Weis, Y. Li, S. Goossens, M. Jung, O. Naji, B. Akesson, N. Wehn, and K. Goossens. *Drampower: Open-Source Dram Power & Energy Estimation Tool*. Accessed: Apr. 7, 2021. [Online]. Available: http://www.drampower.info

[50] N. Corporation. *Nvidia Tensorrt Deep Learning Library and Inference Optimizer*. Accessed: Apr. 7, 2021. [Online]. Available: https://developer.nvidia.com/tensorrt

[51] P. Meloni, D. Loi, G. Deriu, M. Carreras, F. Conti, A. Capotondi, and D. Rossi, "Exploring NEURAghe: A customizable template for APSoC-based CNN inference at the edge," *IEEE Embedded Syst. Lett.*, vol. 12, no. 2, pp. 62–65, Jun. 2020.
[52] *An531: Reducing Power With Hardware Accelerators*, Intel, Santa Clara, CA, USA, 2008.
[53] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. ICLR*, San Diego, CA, USA, 2015. [Online]. Available: https://iclr.cc/archive/www/doku.php%3Fid=iclr2015:main.html#may_8_workshop_poster_session
[54] H. Benmeziane, K. El Maghraoui, H. Ouarnoughi, S. Niar, M. Wistuba, and N. Wang, "A comprehensive survey on hardware-aware neural architecture search," 2021, *arXiv:2101.09336*. [Online]. Available: http://arxiv.org/abs/2101.09336
[55] E. Dilettoso, S. Rizzo, and N. Salerno, "A weakly Pareto compliant quality indicator," *Math. Comput. Appl.*, vol. 22, no. 1, p. 25, 2017.
[56] E. Zitzler, K. Deb, and L. Thiele, *Comparison of Multiobjective Evolutionary Algorithms: Empirical Results*. vol. 8, no. 2. Cambridge, MA, USA: MIT Press, 2000, doi: 10.1162/106365600568202.

PAOLA BUSIA received the B.S. and M.S. degrees in electronics engineering from the University of Cagliari, Italy, in 2017 and 2019, respectively, where she is currently pursuing the Ph.D. degree in electronic and computer engineering. Her research interest includes optimization and deployment of CNNs on resource-constrained systems.

SVETLANA MINAKOVA received the B.Sc. and M.Sc. degrees in computer science from Bauman Moscow State University, Moscow, Russia, in 2015 and 2017, respectively. She is currently pursuing the Ph.D. degree with the LIACS, Leiden University, The Netherlands. She performs research and development in areas of deep learning (DL) and embedded systems and software. Her research and developments activities are a part of European project, called ALOHA—a software framework for runtime-adaptive and secure deep learning on heterogeneous architectures.

TODOR STEFANOV (Member, IEEE) received the Dipl.Ing. and M.S. degrees in computer engineering from Technical University, Sofia, Bulgaria, in 1998, and the Ph.D. degree in computer science from Leiden University, The Netherlands, in 2004. Currently, he is an Associate Professor with Leiden Institute of Advanced Computer Science, Leiden University, and the Head of Leiden Embedded Research Center (LERC) which is a medium-size research group with a strong track record in the area of system-level modeling and synthesis, programming, and implementation of heterogeneous embedded systems. He coauthored more than 80 scientific papers. His research interests include several aspects of embedded systems design, with particular emphasis on system-level design automation, multiprocessor systems-on-chip design, and hardware/software co-design. He was a recipient of the prestigious the 2009 IEEE TCAD Donald O. Pederson Best Paper Award for his journal article "Systematic and Automated Multi-processor System Design, Programming, and Implementation" published in the IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD). He is an Editorial Board Member of the *Journal on Embedded Systems* (Springer). He has also been an Editorial Board Member of the *International Journal of Reconfigurable Computing* and a Guest Associate Editor of *ACM Transactions on Embedded Computing Systems*, in 2013. He has been the General Chair of ESTIMedia 2015 and the Local Organization Co-Chair of ESWeek 2015. Moreover, he serves (has served) on the organizational committees for several leading conferences, symposia, and workshops, such as DATE, ACM/IEEE CODES+ISSS, RTSS, IEEE ICCD, IEEE/IFIP VLSI-SoC, ESTIMedia, SAMOS (as a TPC Member), and IEEE ESTIMedia and ACM SCOPES (as the Program Chair).

LUIGI RAFFO has been a Full Professor of electronics with the University of Cagliari, Italy, since 2006. In 1994, he joined the Department of Electrical and Electronic Engineering, University of Cagliari. He teaches courses on system design, digital and analog electronics design, and processor architectures. Since 2012, he has been the Rector's Delegate for the international research projects. He has been a Coordinator of the Course of Studies in Biomedical Engineering, from 2006 to 2012 and from 2017 to 2018. His research interests include the study, design, development of systems and micro-systems for application where high performance, high-efficiency, and low-power are required. In such a field, he is the author of more than 200 scientific papers.

PAOLO MELONI (Member, IEEE) has been an Assistant Professor at the University of Cagliari, since 2012. His research activity is on the development of advanced digital systems, on the application-driven design and programming of multi-core on-chip architectures, and FPGAs. He is the author of a significant track of international research papers. He teaches advanced embedded systems at the University of Cagliari. He is currently a Scientific Coordinator of the ALOHA (www.aloha-h2020.eu) H2020 project.