Architecture-aware design and implementation of CNN algorithms for embedded inference: the ALOHA project

Paolo Meloni*, Daniela Loi*, Gianfranco Deriu*, Andy D. Pimentel[†], Dolly Sapra[†], Maura Pintor[‡], Battista Biggio[‡], Oscar Ripolles[§], David Solans[§], Francesco Conti[¶], Luca Benini[¶], Todor Stefanov[||], Svetlana Minakova[||], Bernhard Moser**, Natalia Shepeleva**, Michael Masin[††], Francesca Palumbo[‡‡], Nikos Fragoulis[x] and Ilias Theodorakopoulos[x]

* Department of Electrical and Electronic Engineering, University of Cagliari, Italy
[†] Institute of Informatics, University of Amsterdam, The Netherlands
[‡] Pluribus One, Italy
[§] CA Technologies, Spain
[¶] Integrated Systems Laboratory, ETH Zurich, Switzerland
[||] Institute of Advanced Computer Science, Leiden University, The Netherlands
** Software Competence Center Hagenberg, Austria
[††] IBM Research - Haifa, Israel
[‡‡] IDEA Lab, University of Sassari, Italy
[x] IRIDA Labs Computer Vision Systems, Greece
* Corresponding author: paolo.meloni@diee.unica.it

***Abstract*—The use of Deep Learning (DL) algorithms is increasingly evolving in many application domains. Despite the rapid growing of algorithm size and complexity, performing DL inference at the edge is becoming a clear trend to cope with low latency, privacy and bandwidth constraints. Nevertheless, traditional implementation on low-energy computing nodes often requires experience-based manual intervention and trial-and-error iterations to get to a functional and effective solution. This work presents a computer-aided design (CAD) support for effective implementation of DL algorithms on embedded systems, aiming at automating different design steps and reducing cost. The proposed tool flow comprises capabilities to consider architecture- and hardware-related variables at very early stages of the development process, from pre-training hyperparameter optimization and algorithm configuration to deployment, and to adequately address security, power efficiency and adaptivity requirements. This paper also presents some preliminary results obtained by the first implementation of the optimization techniques supported by the tool flow.**

I. Introduction

In recent years, Deep Learning (DL) algorithms have become an extremely promising instrument in the machine learning and artificial intelligence landscape, empowering innovation in a wide variety of application domains from computer vision to speech recognition and automotive systems [1].

Recent trends push towards deployment of DL algorithms on edge nodes as close as possible to the data sources. This approach helps overcoming limitations of cloud-based computing, when it comes to latency, communication bandwidth, privacy, security and reliability. However, performing highly accurate and reliable inference tasks at the edge without compromising performance and energy consumption is still a challenge [2]. A wide landscape of novel very power- and performance-efficient parallel processing architectures are emerging on the market and in literature to meet this need. They are often endowed with accelerators and specialized hardware for speeding-up the most computation-intensive tasks and reducing power consumption, such as convolution layers in Convolutional Neural Networks (CNNs). Two successful examples are the Google Tensor Processing Unit [3] and the NVIDIA Deep Learning Accelerator [4]. Other approaches rely on embedded heterogeneous system-on-chips (SoCs) integrating multi-core processor and using field-programmable gate arrays (FPGAs) optimized for low power operation, such as Xilinx Zynq [5] and Intel Arria 10 [6].

Unfortunately, programming these embedded computing architectures to perform inference task on the edge is highly time-consuming and requires expert developers. In traditional design flows, DL algorithms are in fact designed and trained to improve accuracy without considering the specific features of the processing platform in charge of executing the inference process. This determines the need for multiple design iterations, potentially leading to long tuning phases. The designer skills and the deep knowledge of the target platform features determine the degree of success of the inference process. This limits the adoption of DL mainly to very big actors in the

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 780788

market that can afford the required development costs.

Thus, a computer-aided design tool capable of assisting software developers in implementing DL algorithms on heterogeneous low-energy processing platforms represents a considerable advancement with respect to the state of the art.

In this paper, we propose a software framework capable of automating the selection of an optimal algorithm configuration and the optimization of its implementation according to the given hardware and architectural constraints. This work is part of the activities of the H2020 ALOHA European project, started in January 2018. The goal is to relief designers from the burden related to implementing inference on embedded systems, as well as open the access to DL also to small-medium enterprises and mid-range software development companies, that may focus on the use-case-related problem at the application level rather than on tedious implementation and porting details. The rest of the paper is organized as follows. Section II presents an overview of the tool flow, providing a description of its main components and tools. Section III describes the preliminary experiments performed.

II. The ALOHA tool flow

The ALOHA tool flow is specifically aimed to help software developers when facing the implementation of DL algorithms on modern heterogeneous platforms. It automates different time-consuming developments steps, including the selection of an optimal algorithm configuration, the optimization of its partitioning and mapping on a target processing platform, and the optimization of power and energy savings during its deployment.

The key inputs and outputs of the tool flow are shown in Figure 1. The tool flow receives a configuration file, application-related constraints involving accuracy, security, performance and power, initial DNN(s), a dataset and hardware architecture/specification files as inputs. It generates as output a partitioned and mapped DNN configuration addressing architecture-awareness, ready to be ported on the target computing platform. The three main steps of the proposed tool flow are described in the following sections. Each step is composed of interacting components that influence each other by exchanging HTTP/REST APIs. The overall tool flow exploits a RESTful Microservice architecture and will be integrated considering all requirements posed by Agile development methodologies.

A. *Tailoring the algorithm to the architecture*

Automation of the algorithm design process is the first step of the tool flow. This is done by exploiting a Design Space Exploration (DSE) engine and a set of evaluation and refinement tools, capable of generating the optimal algorithm configuration considering the target task, the constraints, and the target embedded system that will execute the inference task. The DSE engine requests evaluation and refinement of an initial DNN to the tools shown on the right-hand side of Figure 1, with respect to different metrics (i.e. accuracy, security, power and performance). If no initial DNN is provided, by

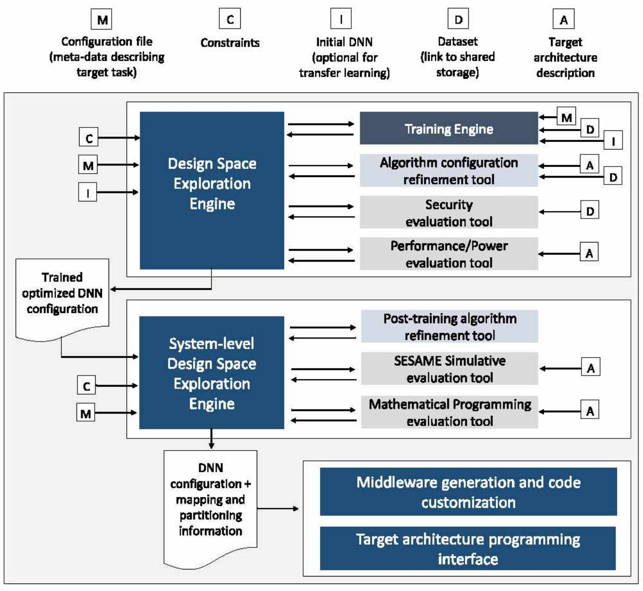
M
Configuration file (meta-data describing target task)
C
Constraints
I
Initial DNN (optional for transfer learning)
D
Dataset (link to shared storage)
A
Target architecture description
Design Space Exploration Engine
Training Engine
Algorithm configuration refinement tool
Security evaluation tool
Performance/Power evaluation tool
Trained optimized DNN configuration
System-level Design Space Exploration Engine
Post-training algorithm refinement tool
SESAME Simulative evaluation tool
Mathematical Programming evaluation tool
DNN configuration + mapping and partitioning information
Middleware generation and code customization
Target architecture programming interface

Fig. 1. General overview of the ALOHA tool flow.

default, the DSE engine generates a population of design points using random or minimum topologies. To reduce the number of evaluations to be performed, the DSE engine uses design-space pruning techniques. However exploration can require several iterations. At each iteration, the DSE creates a Pareto graph populated with design points corresponding to candidate algorithm configurations. When the exploration is finished, the DSE engine triggers the next step of the tool flow.

a) Accuracy evaluation tool: This tool evaluates the accuracy of a candidate algorithm configuration. It is based on a training engine able to support training from scratch or to apply transfer learning to reuse pre-trained networks in a different use-case. The output of the training engine comprises numerical values for the network parameters (weights and bias; hyper parameters) and some meta-information describing accuracy results.

b) Algorithm refinement tool: This tool tries to reduce the computing effort and the energetic cost of the execution of inference of a candidate design point. It applies quantization and pruning methods to the DNN model provided by the DSE. Quantization reduces data precision, using different numerical representation formats in activations and weights. This is needed to lower the data representation from the one used for the original floating-point training to one which allows for parsimonious inference on the target embedded device. Pruning removes low-impact connections between network layers. It includes both iterative method [8] and INQ pruning [9]. The output of this tool is a modified algorithm description that gives the needed accuracy results while reducing the computational workloads.

c) Security evaluation tool: This tool receives as inputs the design point proposed by the DSE engine and the dataset. It then generates an adversarial perturbation that, when applied to the data point, maximizes its probability of being misclassified by the DNN model under evaluation [15]. The output of this tool is a measure of the DNN robustness to adversarial input perturbations, expressed as a security level which can be low, medium or high.

d) Performance/Power evaluation tool: This tool evaluates the performance and the power consumption associated with the execution of the inference of a candidate design point on the target architecture. As shown in Figure 1, it receives as inputs one or several DNN models coming from the DSE engine, and the target architecture description. For every DNN model, the tool generates as output the DNN inference execution time in seconds (Performance), the DNN inference energy consumption in joules (Energy), the number of processing elements prospectively usable for DNN inference (Processors), and the memory required for DNN inference in bytes (Memory).

B. Identifying partitioning and mapping

The second step of the tool flow aims at automating a system-level design process, optimizing the partitioning and the mapping of the algorithm configuration generated by step 1 on the target processing platform. Similarly to the previous step, the design process is driven by a System-level DSE engine. This component controls the exploration of the design space exposed by different partitioning and mappings of the different inference software tasks, and creates a Pareto graph populated with design points corresponding to candidate system-level configurations. To populate the mentioned Pareto graph, the system-level DSE engine requests evaluation of the design points to two evaluation tools: Sesame [10] and Architecture Optimization Workbench (AOW) [11]. To find more efficient mappings of DNN actors to the underlying platform architecture and to optimize the usage of the available resources in the target architecture, the system-level DSE engine may also deploy transformations on the DNN algorithm graph by, for example, merging or splitting actors (i.e., increasing or decreasing the concurrency in the DNN algorithm). Alternatively, it may also invoke the post-training algorithm refinement for parsimonious inference to achieve a workload reduction by considering specific features of the target architecture.

a) Sesame and AOW evaluation tools: In Step 2, the synergy between Sesame and AOW is exploited. AOW explores the whole design space, subject to system requirements and resource constraints (e.g., serializing processing cores and communication buses) using coarse-grain models for computation and communication, while Sesame performs more precise simulation of both computation and communication over a more limited search space for better mapping.

b) Post-training algorithm refinement tool: This tool reduces the computation burden needed for implementing inference by performing a post-selection refinement of the candidate DNN. It is able to apply both a sophisticated on-line data-dependent kernel/component pruning mechanism [14] and a conversion from static to dynamic computing graph to the underlying DNN model. If the process can converge to a solution that delivers a more parsimonious inference, retaining at the same time the accuracy of the initial model within specified margins, the post-training refinement tool generates as output a modified model, otherwise notifies the system-level DSE engine to proceed with the initial trained model.

C. Porting on the target architecture

The last step of the tool flow aims at automating the porting of the target inference application on the target architecture, translating mapping information in adequate calls to computing and communication primitives exposed by the architecture. This step exploits also the power- and performance-related knobs exposed by the platform (VFS, power and clock gating etc.).

III. Preliminary experimental results

A first set of preliminary experiments has been performed to illustrate the potential of the optimization techniques implemented in the ALOHA components.

A. Selection of algorithm parameters

As a first glimpse on ALOHA capabilities, we assessed the possibility of improving computing efficiency of an inference process by considering the specific features of a target architecture when selecting the algorithm-level parameters. To this aim we compare here the performance achievable when executing the well known VGG-16 algorithm on the NEURAghe platform [7] with a custom VGG-like algorithm, modified increasing the number of convolution kernels executed in each layer to match the size of the multiply-and-accumulate (MAC) matrix inside the accelerator. As may me noticed in Table I the architecture-aware configuration better exploits the accelerator, allowing for accuracy improvement.

TABLE I
Original vs custom over-dimensioned VGG-16 algorithm configuration

Benchmark	Performance (GOps/s)	Accuracy (Top-1)
VGG16	172.67	88.4%
NEURAghe-aware VGG16	182.43	89.6%

Moreover, we report in Table II performance levels achievable on the same architecture using iterative quantization/retraining to change the data format used to represent activation and weights. A first optimization can be achieved considering the possibility of NEURAghe to operate on 16 and 8 bits data formats. The accelerator uses the same MAC hardware actor executing 16 bits operation to execute two different 8 bits operation. Thus this bring to a significant speed-up that can be captured by the toolflow and exploited when allowed by the specific use case (see third row). Quantization can be used when needed as a compression method, to reduce pressure on memory bandwidth, often stressed when weights are loaded from DDR to on-chip memory, as may be noticed when comparing the two configurations with 16-bit activations. The iterative re-training procedure after quantization was capable of reconstructing accuracy even for reduced precisions, stabilizing cumulative loss without significant degradation.

TABLE II
NEURAGHE PERFORMANCE ON DIFFERENTLY QUANTIZED VGG-16 CONFIGURATIONS

Quantization	Performance
16-bit activations 16-bit weights	*172 GOPs/s*
16-bit activations 8-bit weights	*175 GOPs/s*
8-bit activations 8-bit weights	*335 GOPs/s*

TABLE III
EFFECTS OF ADVERSARIAL TRAINING ON THE CNN ROBUSTNESS TO NOISE-BASED ATTACKS

ε	Classification accuracy	
max	*Without adversarial training*	*With adversarial training*
0	80%	80%
0.4	20%	75%
0.8	10%	60%
1	9%	45%

B. Evaluation of security against adversarial attacks

In this section we present an experiment related with evaluation and improvement of the security level of a deep network, taking into account handwritten digit classification as a use-case. For this experiment we used a task-specific CNN model of Keras library [12]. We trained the underlying model on the MNIST dataset, after normalizing all images in $[0, 1]$ by dividing the pixel values by 255, and manipulated $10,000$ test samples using the Fast Gradient Sign Method (FGSM) attack algorithm [13]. This attack bounds the max-norm distance between a (legitimate) input $\mathbf{x}$ and its adversarial counterpart $\mathbf{x}'$ as $\|\mathbf{x} - \mathbf{x}'\|_\infty \leq \varepsilon$. Thus, every pixel p in the image $\mathbf{x}'$ is manipulated independently in the interval $[p - \varepsilon, p + \varepsilon]$. An example of manipulated MNIST handwritten digit is shown in Figure 2. Note that, within this setting, the adversarial perturbation is almost imperceptible to the human eye, though still effective to mislead recognition.

We then applied a defense mechanisms, called *adversarial training* [13], by augmenting the training dataset with adversarial examples and by re-training the neural network and repeating the FGSM attack for $\varepsilon \in \{0, 0.4, 0.8, 1\}$ against the robust network. The effect of the adversarial training is analyzed by gradually changing perturbation ε. In both cases, as expectable, classification accuracy degrades under attacks characterized by an increasing perturbation. However, when using adversarial training it was possible to significantly increase resilience to attacks, as reported in Table III for different noise levels.

CONCLUSIONS

In this paper, we have introduced a CAD tool flow to overcome the limits of traditional practices currently used to

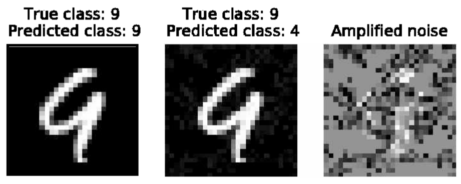

Fig. 2. An example of manipulated MNIST handwritten digits that mislead classification by a CNN, crafted with the FGSM attack algorithm [13] with $\varepsilon = 0.05$.

deploy deep learning algorithms at the edge. In contrast to previous approaches, we have focused on the possibility of automating the selection of an optimal algorithm configuration and the optimization of its implementation, considering the specific features of the processing platform executing the inference task during the whole development process. We have evaluated the potential of the optimization techniques implemented in the proposed tool flow and presented first results.

Acknowledgments: The authors would like to thank all the partners involved in the ALOHA project for their contribution.

REFERENCES

[1] I. Goodfellow, Y. Bengio, and A. Courville, "Deep Learning," MIT Press, 2016, http://www.deeplearningbook.org

[2] X. Xu, Y. Ding, S. Hu, M. Niemier, J. Cong, Y. Hu and Y. Shi, "Scaling for edge inference of deep neural networks," Nature Electronics, vol. 1, pp. 216-222, April 2018.

[3] N. P. Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit," Proceedings of the 44th Annual International Symposium on Computer Architecture, pp. 1–12, June 2017.

[4] Nvidia. 2018. NVDIA Deep Learning Accelerator http://nvdla.org/

[5] Xilinx. 2018. Zynq-7000 SoC https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

[6] Intel. 2018. Intel Arria 10 FPGAs and SoCs. https://www.intel.com/content/www/us/en/products.html

[7] 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," 2017, https://arxiv.org/abs/1712.00994

[8] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv and Y. Bengio "Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations," 2016, http://arxiv.org/abs/1609.07061

[9] A. Zhou, A. Yao, Y. Guo, L. Xu and Y.Chen "Incremental Network Quantization: Towards Lossless CNNs with Low-Precision Weights," 2017, https://arxiv.org/abs/1702.03044

[10] A. D. Pimentel, C. Erbas and S. Polstra "A systematic approach to exploring embedded system architectures at multiple abstraction levels," IEEE Transactions on Computers, vol. 55, pp. 99–112, 2006.

[11] M. Masin, L. Limonad, A.Sela, D. Boaz, L. Greenberg, N. Mashkif and R. Rinat. "Pluggable Analysis Viewpoints for Design Space Exploration," Procedia Computer Science, vol. 16, pp. 226–235, 2013.

[12] Keras. 2018. https://github.com/keras-team/keras/blob/master/examples/mnist$_{c}nn.py$

[13] I. J. Goodfellow, J. Shlens and C. Szegedy "Pluggable Analysis Viewpoints for Design Space Exploration," 2015, https://arxiv.org/abs/1412.6572

[14] I. Theodorakopoulos, V. Pothos, D. Kastaniotis and N. Fragoulis "Parsimonious Inference on Convolutional Neural Networks: Learning and applying on-line kernel activation rules," 2017, https://arxiv.org/abs/1701.05221

[15] B. Biggio and F. Roli "Wild Patterns: Ten Years After the Rise of Adversarial Machine Learning," Pattern Recognition, vol. 84, 2018, pp. 317-331.