Optimization and deployment of CNNs at the edge: the ALOHA experience

Invited Paper

Paolo Meloni
Daniela Loi
Paola Busia
Gianfranco Deriu
University of Cagliari, Italy
paolo.meloni@diee.unica.it
daniela.loi@diee.unica.it
paola.busia@unica.it
gianfranco.deriu@unica.it

Andy D. Pimentel
Dolly Sapra
University of Amsterdam,
The Netherlands
a.d.pimentel@uva.nl
d.sapra@uva.nl

Todor Stefanov
Svetlana Minakova
Leiden University,
The Netherlands
t.p.stefanov@liacs.leidenuniv.nl
s.minakova@liacs.leidenuniv.nl

Francesco Conti
Luca Benini
ETH Zurich, Switzerland
fconti@iis.ee.ethz.ch
lbenini@iis.ee.ethz.ch

Maura Pintor
Battista Biggio
Pluribus One, Italy
maura.pintor@pluribus-one.it
battista.biggio@pluribus-one.it

Bernhard Moser
Natalia Shepeleva
SCCH, Austria
bernhard.moser@scch.at
natalia.shepeleva@scch.at

Nikos Fragoulis
Ilias Theodorakopoulos
Irida Labs, Greece
nfrag@iridalabs.gr
iltheodorako@iridalabs.gr

Michael Masin
IBM Research, Israel
michaelm@il.ibm.com

Francesca Palumbo
University of Sassari, Italy
fpalumbo@uniss.it

ABSTRACT

Deep learning (DL) algorithms have already proved their effectiveness on a wide variety of application domains, including speech recognition, natural language processing, and image classification. To foster their pervasive adoption in applications where low latency, privacy issues and data bandwidth are paramount, the current trend is to perform inference tasks at the edge. This requires deployment of DL algorithms on low-energy and resource-constrained computing nodes, often heterogenous and parallel, that are usually more complex to program and to manage without adequate support and experience. In this paper, we present ALOHA, an integrated tool flow that tries to facilitate the design of DL applications and their porting on embedded heterogenous architectures. The proposed tool flow aims at automating different design steps and reducing development costs. ALOHA considers hardware-related variables and security, power efficiency, and adaptivity aspects during the whole development process, from pre-training hyperparameter optimization and algorithm configuration to deployment.

CF '19, April 30-May 2, 2019, Alghero, Italy

ACM ISBN 978-1-4503-6685-4/19/05…$15.00
https://doi.org/10.1145/3310273.3323435

CCS CONCEPTS

• **Computer systems organization → Embedded systems**; • **Hardware → Hardware accelerators**.

KEYWORDS

Convolution Neural Networks, FPGAs, Hardware accelerators

ACM Reference Format:
Paolo Meloni, Daniela Loi, Paola Busia, Gianfranco Deriu, Andy D. Pimentel, Dolly Sapra, Todor Stefanov, Svetlana Minakova, Francesco Conti, Luca Benini, Maura Pintor, Battista Biggio, Bernhard Moser, Natalia Shepeleva, Nikos Fragoulis, Ilias Theodorakopoulos, Michael Masin, and Francesca Palumbo. 2019. Optimization and deployment of CNNs at the edge: the ALOHA experience: Invited Paper. In *Proceedings of the 16th conference on Computing Frontiers (CF '19), April 30-May 2, 2019, Alghero, Italy*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3310273.3323435

1 INTRODUCTION

Deep Learning (DL) algorithms are at the forefront of Artificial Intelligence (AI) techniques [7]. Ever-growing research and implementation efforts are dedicated to the development of novel algorithm configurations able to increase prediction and classification accuracy of always more advanced DL applications. However, improvement in performance comes at the cost of a higher

computational complexity and additional memory and processing power requirements [15]. While this excessive resource demand is not a problem for the training phase that is typically executed on high-performance computing facilities, it can represent an issue for exploiting the emerging Edge Computing paradigm [13], which calls for deploying the inference task on embedded devices with limited area and power budget.

Deploying the inference task of DL models at the edge has become a popular trend in particular for those applications where power, privacy, bandwidth, latency, and reliability are critical constraints. In order to cope with the increasing need for computational power and, at the same time, with the limited energy budget, modern mobile systems are based on complex AI-optimized chipset architectures featuring heterogeneous computing devices, including Graphical Processing Units (GPUs), Application-Specific Integrated Circuits (ASICs), Neural Network Processing Units (NNPUs), multicore processors and Field-Programmable Gate Arrays (FPGAs). In these devices, different processing cores can cooperate by executing different computing kernels, maximizing performance and reducing power consumption [10, 17].

Despite the variety of resources available, optimizing these heterogeneous computing architectures for performing low-latency and energy-efficient DL inference tasks without compromising performance is still a challenge [5]. This mainly because productivity in DL algorithm development is typically reduced by a deep-rooted dichotomy: the design and training of the learning algorithm and the inference of the resulting model are two distinct activities. The main problem with the current DL solutions development flow is that the training phase leads to the selection of an optimal algorithm configuration mainly considering accuracy as the only main design objective. The selected algorithm configuration has little or no correspondence with the specific features of the processing hardware architecture in charge of executing the inference task. Developers of hardware-software systems dealing with the inference process commonly start from pre-trained networks, trying to optimize as much as possible their execution on the target computing platform. This dichotomy determines the need for multiple design iterations, potentially leading to long tuning phases, overloading designers and with results highly depending on their skills.

In this paper, we present ALOHA, an integrated tool flow that tries to make the design of DL applications and their porting on embedded heterogenous architectures as simple and painless as possible. The proposed tool flow aims at automating different design steps and reducing development costs by bridging the gap between DL algorithm training and inference phases. The tool considers hardware-related variables and security, power efficiency, and adaptivity aspects during the whole development process, from pre-training hyperparameter optimization and algorithm configuration to deployment.

2 ALOHA FRAMEWORK

The overall ALOHA framework is shown in Figure 1. It essentially automates three different steps: algorithm selection, application partitioning and mapping, and deployment on target hardware. Through the user interface, the tool flow receives as inputs the following data: a starting neural network, a dataset, a target architecture description, a configuration file containing information about the target application, and a set of constraints on accuracy, security, performance and power that the application must satisfy. Starting from this set of user-specified inputs, the tool flow generates a partitioned and mapped neural network configuration, ready to be ported on the target processing architecture, that co-optimizes both the application-level accuracy and the required security level, inference execution time and power consumption.

A RESTful microservices approach is used to structuring the proposed tool flow architecture implementing an Agile development methodology. As shown in Figure 1, each step of the development flow is broken into smaller, completely independent components, which interact and influence each other by exchanging HTTP request/response API (Application Programming Interface) calls. Docker containers are used to create loosely isolated running environments, so that each component can be independently built and deployed to implement a specific feature of the ALOHA tool. All the components have access to a shared storage image containing a MongoDB database. Deployments of the different components are managed through a container orchestration platform. The standard ONNx (Open Neural Network Exchange) is used for exchanging deep learning models between the different tool flow components.

The main role of the user interface is to provide ease-of-use and accessibility to the tool flow, ensuring adoption among deep learning practitioners. The graphic interface guides the user during the definition of a use-case, shows work in progress and provides visualization of results on the implemented tasks in the form of graphs and tables.

In the following sections, we provide full details on all the three steps required in the ALOHA tool flow and a description of the developed user interface.

2.1 Tool flow environment

This section describes the three steps of the proposed tool flow.

Step 1 - Automation of the algorithm design process. The first step is guided by a Design Space Exploration (DSE) engine, which performs a multi-objective exploration using a genetic algorithm and accesses a set of refinement and evaluation tools to generate the optimal DNN configuration, depending on the target application, the constraints imposed, and the target hardware platform selected by the user (see Figure 1). Each evaluation tool is able to estimate a specific parameter for each design point. The DSE engine starts its operation using an initial DNN configuration provided as input by the user, or generating itself a first population of design points corresponding to random algorithm configurations in terms of number of layers, kernel size, layer connectivity. Then, to refine and evaluate the generated design points the DSE engine cooperates with the following set of satellite tools:

- Refinement tool for parsimonius inference. It tries to reduce the computational workload associated with the inference execution of a candidate design point, the size of the memory footprint, and the IO bandwidth requirements, by applying quantization and pruning techniques;
- Accuracy evaluation tool. It assesses the accuracy of a candidate design point. This tool is based on a training engine,

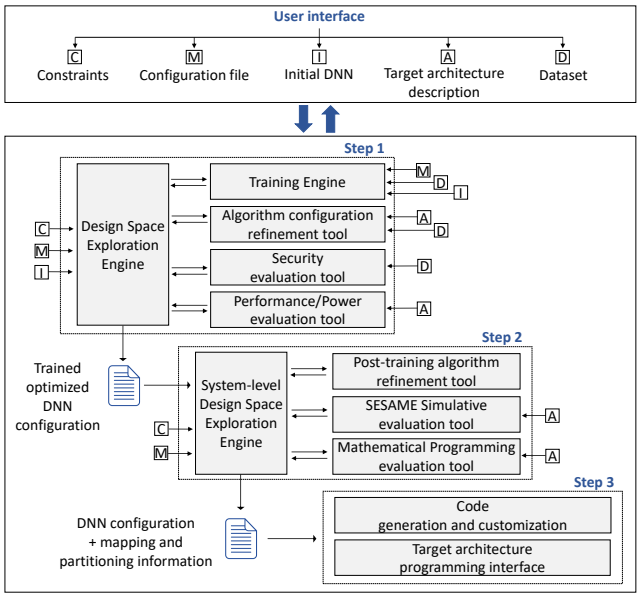

Figure 1: General architecture of the ALOHA software framework. Nodes in the upper part of the figure represent the key inputs of the tool flow specified by the users.

which can be configured to train a DNN model from scratch or to apply transfer learning techniques to it. In the latter mode, the engine tries to reuse pre-trained network models, personalizing them for the target use case. The accuracy evaluation tool offers local hyper-parameter exploration, flexible data parsing (multiple input formats) and flexible use-case configuration (multiple AI tasks, such as classification, detection, tracking).

- Security evaluation tool. It evaluates the robustness of a candidate design point to adversarial attacks happening in prediction phase. Adversarial attacks are testing data that are maliciously altered to be misclassified by a candidate DNN model [2, 3]. These attacks are generated simulating the worst case, by algorithmically computing the modifications which maximize the error rate of the DNN.
- Performance and Power evaluation tool. It evaluates the performances and the power consumption associated with the execution of the inference of a candidate design point on a target architecture. It converts the DNN model coming from the DSE engine to an analyzable application model (i.e. a Cyclo-Static DataFlow model). Taking into consideration the target architecture description received as input, this satellite tool allows to pre-estimate what will be the execution time and power consumption of the actors inside the generated model.

When the exploration is finished and the most suitable design choice is identified, the DSE engine triggers the next step of the tool flow.

Step 2 - Optimization of the partitioning and mapping. The second step of the tool flow is guided by a System-level DSE engine, which identifies the best partitioning of the algorithm configuration generated by step 1 in sub-tasks, and finds the optimal mapping scheme of these sub-tasks on the different processing units available in the target hardware platform, able to satisfy requirements and constraints specified by the user (i.e. throughput, latency and power). Similarly to the previous step, this is done using a genetic algorithm for surfing the design space and requiring evaluation of the candidate partitioning and mapping scheme to two satellite tools: Sesame [11] and Architecture Optimization Workbench (AOW) [8]. These tools simulate computation and task-to-task communication and provide approximates on execution (cycle) times, energy consumption, hardware utilization and resource contention. AOW explores the whole design space using an analytic approach, while Sesame performs more precise simulation over a more limited search space.

The System-level DSE engine can also deploy transformations on the DNN algorithm graph by, for example, merging or splitting tasks (i.e., increasing or decreasing the concurrency in the DNN algorithm), to find more efficient mappings of sub-tasks, to optimize the usage of the available hardware resources and to adapt to different operating modes [1].

Alternatively, it can access the post-training algorithm refinement tool for parsimonious inference to achieve a workload reduction by considering specific features of the target architecture (see Figure 1). This satellite tool tries to reduce the computational workload by applying both a sophisticated on-line data-dependent kernel/component pruning mechanism [16] and a conversion from static to dynamic computing graph to the DNN model. The pruning process is performed by gradually weakening the contribution of unnecessary kernels and layers in the CNN architecture. In the dynamic pruning, several components (i.e. convolutional kernels, groups of kernels, layers etc.) are conditionally executed according to learned rules, and based on the respective data being processed, by the means of special, trainable processing modules called LKAMs (Learning Kernel Activation Modules). These modules are capable to switch on and off individual kernels of any layer, depending on its input, which is the output of the previous convolutional layer.

If the execution of the self-pruning processes allow to deliver a more parsimonious inference while simultaneously ensuring the accuracy of the initial model within specified margins, the post-training refinement tool generates as output a modified DNN model, otherwise notifies the system-level DSE engine to proceed with the initial trained model.

When the system-level exploration is finished and the more efficient mapping of the DNN configuration is identified, the last step of the tool flow can be carried out.

Step 3 - Automation of the porting process. The last step of the tool flow is the porting of the DNN configuration on the target hardware architecture. A programming interface receives as input the partitioning and mapping information generated by step 2, and translates them into specific calls to computing and communication primitives exposed by the target hardware architecture. The generated platform-specif code is then customized to reduce as much as possible power consumption and improve performance using, when possible, optimization techniques such as power gating, clock gating and frequency scaling.

2.2 User interface design

The ALOHA user interface provides users with the ability to monitor the state of each step of multiple projects through the tool flow at any time. It is built using a JavaScript-based Kanban board within React framework. As shown in Figure 2, the Kanban approach allows to have a simple and intuitive graphic interface, with only the important information highlighted.

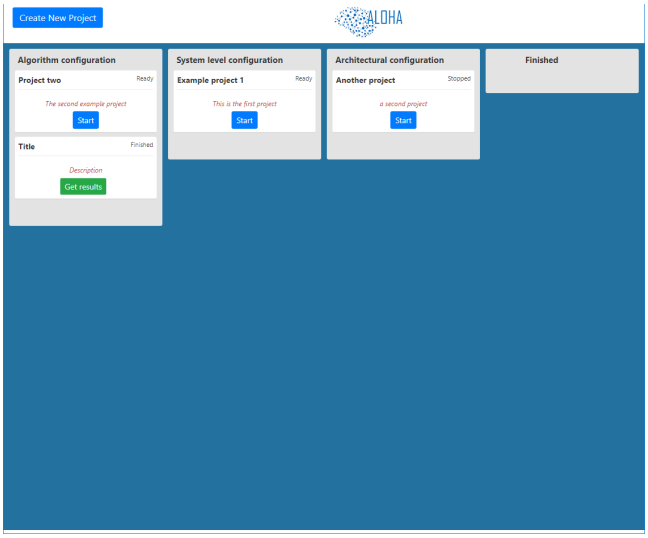

Figure 2: ALOHA user interface.

The visual board is divided into three columns, each of them representing a specific development step of the tool flow process. Each column is populated with cards that represent work items of different types. Cards can be added in a column, deleted, or dragged from one column to the other. Each card goes moves from left to right until the full workflow is completed and the work is done. To implement a new use-case, the user starts creating and configuring a new project. This is done by:

- editing a project title and a description;
- uploading an architecture specification file, which provide information about the target hardware platforms in terms of population of computing elements, connectivity, and available operating modes (i.e. data types, working frequency and gating conditions).
- designating the desired values for accuracy, security, execution time and power consumption;
- specifying a link to the shared folder location that stores the dataset to be used;
- uploading an initial DNN configuration;
- specifying information about the target application to be implemented (i.e. task type, training rate, number of training epochs, batch size, optimization method, loss function).

3 EXPERIMENTAL RESULTS

In this section we present the capabilites of some of the utilities involved in the ALOHA toolflow. Namely, we show:

- an evaluation of a CNN with respect to the possibility of applying parsimonious inference, during algorithm selection, see Section 3.1;
- an evaluation of a CNN with respect to its resilience to adversarial attacks, see Section 3.2;
- the effectiveness of applying run-time parsimonious inference, to reduce the computational burden and to speed-up the CNN inference on one of the project reference processing platforms, see Section 3.3;

All the presented results are obtained using the toolflow instruments on state-of-the-art network topologies and image benchmarks.

3.1 Algorithm refinement for parsimonious inference

To preliminary evaluate the functionality of the algorithm configuration refinement tool envisioned in Step 1, we iteratively fine-tuned a MobilNet topology to lower precision using a quantization/re-training scheme based on the PArameterized Clipping acTivation (PACT) function [4]. We used a pre-trained full-precision version of MobilNet (on CIFAR-10) as input, and we applied the following iterative approach:

(1) fine-tuning the current version of the network;
(2) evaluating its convergence rate to stop fine-tuning when convergence is flat;
(3) lowering the precision of the data representation.

Figure 3 shows that both weights and activations can be quantized to 4-bits of precision while still achieving accuracy comparable to a full precision MobileNet network. Starting directly from 8 bits and lowering to 4 bits, the quality drop in terms of accuracy is below 5% within around 280 epochs.

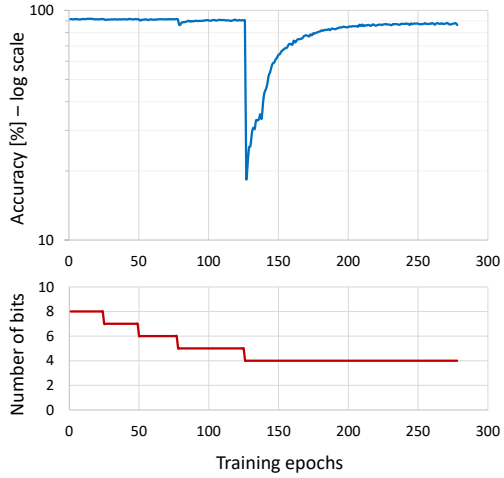

Figure 3: Evolution of the accuracy of the model for different resolutions.

3.2 CNN robustness to adversarial attacks

To evaluate the capability of the tool flow to assess the security of a candidate design point to adversarial examples, we considered a pre-trained CNN on the CIFAR 10 dataset (93.78% accuracy).[1] We manipulated 1000 test samples with the Iterative Fast Gradient Sign Method (I-FGSM) attack algorithm, using 20 iterations [6]. This attack bounds the max-norm distance between the source image $\mathbf{x}$ and its adversarial counterpart $\mathbf{x}'$ as $\|\mathbf{x} - \mathbf{x}'\|_\infty \leq \varepsilon$. This amounts to manipulating each pixel p in $\mathbf{x}'$ independently in the interval $[p - \varepsilon, p + \varepsilon]$. We run this attack for 100 different values of perturbation ε. In Figure 4 we show some adversarial examples obtained with low ($\varepsilon = 0.03$) and high ($\varepsilon = 0.1$) perturbation.

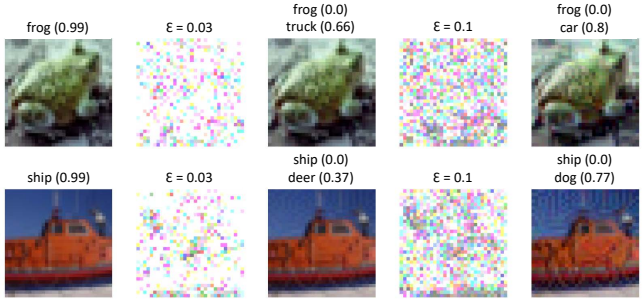

Figure 4: Examples of adversarial attacks with $\varepsilon = 0.03$ (low perturbation) and $\varepsilon = 0.1$ (high perturbation). The noise masks are magnified for visibility purposes.

In Figure 5 we report the results of the complete security evaluation procedure, showing how accuracy decreases as the perturbation ε increases, originally by considering 1,000 images (green line, *slow evaluation*). Considering that this evaluation can be computationally demanding (it required 1,843 seconds to complete on a workstation with 128GB RAM and a GPU NVIDIA Quadro M6000 24GB), we performed a second evaluation to evaluate the usability inside an iterative DSE process, as envisioned in ALOHA. To reduce the computational cost, we simply used 100 samples (instead of 1,000) and 20 values for ε (instead of 100) to estimate the security evaluation curve (red line, *fast evaluation*). The trend is equivalent to the previous experiment, but required only 35 seconds, suggesting the possibility of using a *faster* security evaluation process when iterative DSE is involved.

3.3 Run-time parsimonious inference effectiveness

To evaluate the effectiveness of the support for run-time reduction of the computational load during inference, we performed two different experiments. In the first experiment, we considered a general image recognition problem utilizing the ImageNet ILSVRC 2012 dataset [12] and we evaluated the proposed post-training parsimonius inference technique on the well known VGG-16 model [14]. This enables to show how this technique can be applied to quite complex and compute intensive network structures and datasets. In the second experiment, to evaluate the achievable benefits, we refer to a simpler algorithm (LeNet [7]) on a resource constrained processing platform, integrating an ARM processor (software implementation) and a FPGA-based CNN accelerator (hardware implementation), and we measure the speed-up when run-time parsimonious inference (PI) is enabled.

[1] https://github.com/aaron-xichen/pytorch-playground

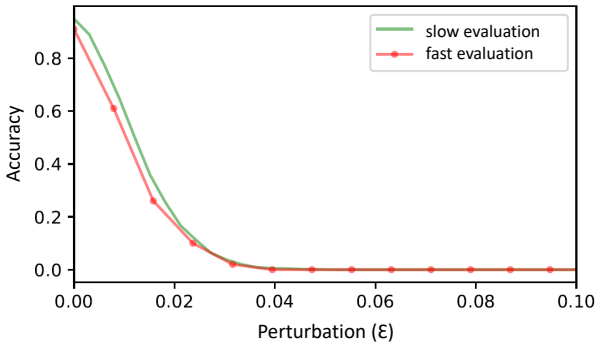

Figure 5: Security evaluation curves computed with 1000 (green line) and 100 (red line) images.

Computational load reduction evaluation. Using the utility integrated in ALOHA, we performed an initial model analysis to identify the demanding nodes and algorithmic components delivering the largest computational gains. Then, we defined a set of specialized hyper-parameters based on the architecture of the VGG-16 model, and we added LKAM modules into all convolutional layers to control the kernels activity. Finally, we performed a specialized post-training refinement process to appropriately refine the model.

LKAM modules identify which filtering kernels can be activated. This is shown in Figure 6, where we report the resulting kernel activity profile for each layer of the trained parsimonious model.

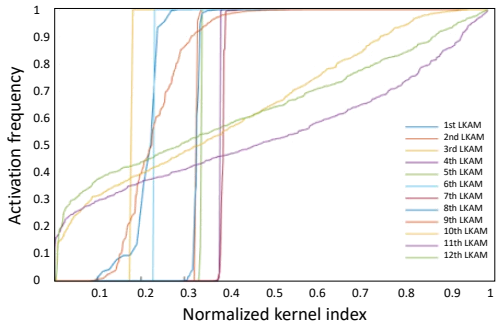

Figure 6: Kernel activity profile.

In the graph, the vertical axis corresponds to the activation frequency of a particular kernel throughout the validation dataset, while each value in the horizontal axis corresponds to a specific kernel in a layer. For visualization purposes, the horizontal axis has been normalized. Each curve represents a layer. A step-like curve implies that kernels in the corresponding layer are either permanently active or inactive, while a smooth curve indicates that most kernels whose operation is data-dependent.

The statistical analysis on the switching activity showed that, on average, only 66.14% of the kernels are active in the layers of the network throughout the target validation dataset. Seven of the convolutional layers are operating in a static or close-to-static mode, enabling the permanent pruning of the redundant kernels from the model. The resulting model achieves a respectable of 70.4% accuracy, presenting a $\sim$ 2% deficit to the reference model, but with an impressive 48.31% reduction in terms of FLOPs.

Inference speed evaluation. We applied LKAM modules to a LeNet-5 network model, pre-trained on CIFAR-10 dataset. The LeNet model implements three convolutional layers, as shown in Table 1, thus we added two LKAM modules linking respectively, Conv1 to Conv2 and Conv2 to Conv3. In Table 2, we report the resulting kernel activity profile on 3 test images.

Table 1: Architecture of the LeNet model.

Layer (name; description)	Image size	Input Features	Output Features	Kernel size
Conv1; Convolution	32x32	3	32	5x5
Conv2; Convolution	16x16	32	32	5x5
Conv3; Convolution	8x8	32	64	5x5

Table 2: Percentage of the number of inactive kernels for each convolutional layer of the target model.

Layer	Test 1	Test 2	Test 3
Conv1	0%	0%	0%
Conv2	31,25%	43,75%	40,63%
Conv3	31,25%	51,56%	40,63%

We then measured the inference time needed to execute the modified model on NEURAghe [9], a Zynq-based processing platform that contains both a dual ARM Cortex A9 processor (667 MHz) and a CNN accelerator implemented in the programmable logic. We performed two different experiments, one using only the ARM processor and one exploiting the accelerator. The results are shown in Table 3 and Table 4.

Table 3: Comparison between the execution time needed to implement the baseline and the modified model on ARM. For each convolutional layer, the execution time in ms and the percentage of speedup after the application of the post-training parsimonious inference is reported.

	Baseline model	Test 1 (t; speedup)	Test 2 (t; speedup)	Test 3 (t; speedup)
Conv2	112	78.69; -30%	64.03; -43%	67.55; -40%
Conv3	56	40.66; -27%	31.28; -44%	35.54; -36%
Total	168	119.35; -29%	95.31; -43%	103.09; -39%

As may be noticed, kernel deactivation results in significant speed-up in both experiments. Parsimonious inference is more effective on pure software execution, since it more directly deactivates operations in the convolution process. When the accelerator is used, even if execution time is significantly decreased, some overhead related with accelerator preparation and activation slightly limits the speed-up due to kernel deactivation.

Table 4: Comparison between the execution time needed to implement the baseline and the modified model on NEURAghe, configured on a Xilinx Zynq Z-7010 SoC. For each convolutional layer, the execution time in ms and the percentage of speedup after the application of the post-training parsimonious inference is reported.

	Baseline model	Test 1 (t; speedup)	Test 2 (t; speedup)	Test 3 (t; speedup)
Conv2	9.4	7.58; -19%	6.34; -33%	6.49; -31%
Conv3	16.97	12.9; -24%	10.82; -36%	11.91; -30%
Total	26,37	20.48; -22%	17.16; -35%	18.4; -30%

4 CONCLUSION

We have presented ALOHA, a software framework for automating the development process of deep learning inference tasks on heterogeneous computing architectures. Our approach relies on considering both hardware-related variables and security, power efficiency, and adaptivity aspects during the whole development process, from pre-training hyperparameter optimization and algorithm configuration to deployment. We have presented the capabilities of the utilities involved in the toolflow.

ACKNOWLEDGMENTS

This work has received funding from the European Union's Horizon 2020 Research and Innovation Programme under grant agreement No. 780788. The authors would like to thank Giuseppe Desoli, Giulio Urlini, Adriano Souza Ribeiro, Werner Kloihofer, Cristina Chesta, Yaniv Ben Zriham and Gilad Wainreb for their support.

REFERENCES

[1] Onur Derin, Emanuele Cannella, Giuseppe Tuveri, Paolo Meloni, Todor Stefanov, Leandro Fiorin, Luigi Raffo, and Mariagiovanna Sami. 2013. A system-level approach to adaptivity and fault-tolerance in NoC-based MPSoCs: The MADNESS project. *Microprocessors and Microsystems* 37, 6-7 (2013), 515–529.

[2] Battista Biggio et al. 2013. Evasion attacks against machine learning at test time. In *Machine Learning and Knowledge Discovery in Databases (ECML PKDD), Part III (LNCS)*, Vol. 8190. Springer Berlin Heidelberg, 387–402.

[3] Christian Szegedy et al. 2014. Intriguing properties of neural networks. In *International Conference on Learning Representations*.

[4] Jungwook Choi et al. 2018. PACT: Parameterized Clipping Activation for Quantized Neural Networks. *CoRR* abs/1805.06085 (2018). arXiv:1805.06085 http://arxiv.org/abs/1805.06085

[5] Xu Xiaowei et al. 2018. Scaling for edge inference of deep neural networks. *Nature Electronics* 1, 4 (Apr 2018), 216–222.

[6] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In *International Conference on Learning Representations*.

[7] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep Learning. *Nature* 521 (2015), 436–444.

[8] Michael Masin, Lio Limonad, Aviad Sela, David Boaz, Lev Greenberg, Nir Mashkif, and Ran Rinat. 2013. Pluggable Analysis Viewpoints for Design Space Exploration. *Procedia Computer Science* 16 (2013), 226–235.

[9] Paolo Meloni, Alessandro Capotondi, Gianfranco Deriu, Michele Brian, Francesco Conti, Davide Rossi, Luigi Raffo, and Luca Benini. 2018. NEURAghe: Exploiting CPU-FPGA Synergies for Efficient and Flexible CNN Inference Acceleration on Zynq SoCs. *ACM Transactions on Reconfigurable Technology and Systems* 11, 3 (2018), 1–24.

[10] Danilo Pani, Paolo Meloni, Giuseppe Tuveri, Francesca Palumbo, Massobrio Paolo, and Luigi Raffo. 2017. An FPGA Platform for Real-Time Simulation of Spiking Neuronal Networks. *Frontiers in Neuroscience* 11 (2017), 90.

[11] Andy D. Pimentel, Cagkan Erbas, and Simon Polstra. 2006. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Trans. Comput.* 55, 2 (Feb 2006), 99–112.

[12] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2014. ImageNet Large Scale Visual Recognition Challenge. *CoRR* abs/1409.0575 (2014). arXiv:1409.0575 http://arxiv.org/abs/1409.0575

[13] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge Computing: Vision and Challenges. *IEEE Internet Things J.* 3, 5 (2016), 637–646.

[14] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large Scale Image Recognition. *CoRR* abs/1409.1556 (2014). http://arxiv.org/abs/1409.1556

[15] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel Emer. 2017. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proc. IEEE* 105, 12 (2017), 2295–2329.

[16] Ilias Theodorakopoulos, V. Pothos, Dimitris Kastaniotis, and Nikos Fragoulis. 2017. Parsimonious Inference on Convolutional Neural Networks: Learning and applying on-line kernel activation rules. *CoRR* abs/1701.05221 (2017). http://arxiv.org/abs/1701.05221

[17] Stylianos I. Venieris, Alexandros Kouris, and Christos-Savvas Bouganis. 2018. Toolflows for Mapping Convolutional Neural Networks on FPGAs: A Survey and Future Directions. *ACM Comput. Surv.* 51, 3, Article 56 (June 2018), 39 pages.