Journal of Signal Processing Systems (2023) 95:1091–1113
https://doi.org/10.1007/s11265-023-01855-x

An Automated Design Flow for Adaptive Neural Network Hardware Accelerators

Francesco Ratto[1] · Ángela Porras Máinez[2] · Carlo Sau[1] · Paolo Meloni[1] · Gianfranco Deriu[1] · Stefano Delucchi[4] · Massimo Massa[4] · Luigi Raffo[1] · Francesca Palumbo[3]

Received: 3 April 2021 / Revised: 13 February 2023 / Accepted: 15 February 2023 / Published online: 26 April 2023

Abstract
Image and video processing are one of the main driving application fields for the latest technology advancement of computing platforms, especially considering the adoption of neural networks for classification purposes. With the advent of Cyber Physical Systems, the design of devices for efficiently executing such applications became more challenging, due to the increase of the requirements to be considered, of the functionalities to be supported, as well as to the demand for adaptivity and connectivity. Heterogeneous computing and design automation are then turning into essential. The former guarantees a variegated set of features under strict constraints (e.g., by adopting hardware acceleration), and the latter limits development time and cost (e.g., by exploiting model-based design). In this context, the literature is still lacking adequate tooling for the design and management of neural network hardware accelerators, which can be adaptable and customizable at runtime according to the user needs. In this work, a novel almost automated toolchain based on the Open Neural Network eXchange format is presented, allowing the user to shape adaptivity right on the network model and to deploy it on a runtime reconfigurable accelerator. As a proof of concept, a Convolutional Neural Network for human/animal classification is adopted to derive a Field Programmable Gate Array accelerator capable of trading execution time for power by changing the resources involved in the computation. The resulting accelerator, when necessary, can consume 30% less power on each layer, taking about overall 8% more time to classify an image.

Keywords Hardware Acceleration · Adaptivity · Design Automation · Convolutional Neural Network · Reconfigurable Computing · High Level Synthesis

1 Introduction

Cyber Physical Systems (CPSs) advanced a lot in the latest years. They are not only capable of strong interactions and information exchange with the environment, but porting at the edge the possibility of taking decisions, bringing Artificial Intelligence (AI) on small embedded platforms, and making them capable of adapting to different environmental and systems stimulus, has pushed their level of autonomy. The support of variable and intensive workloads, often requiring real-time execution, and the increasing need of addressing several concurrent functional and non-functional requirements, led designers to adopt heterogeneous platforms integrating different types of SoftWare (SW) and HardWare (HW) resources, as different types of cores, application specific units, and configurable logic. Recent trends have seen Field Programmable Gate Array (FPGA) devices, traditionally employed for rapid prototyping and low-volume application purposes, gaining momentum in production (e.g., Lattice Semiconductor FPGA in edge devices [1]) since they can guarantee HW acceleration, execution flexibility, and energy efficiency.

In the context of modern smart CPSs, we have been involved in the ECSEL project FitOptiVis (From the cloud to the edge - smart IntegraTion and OPtimisation Technologies for highly efficient Image and VIdeo processing Systems) [2, 3]. The project has studied novel design and

✉ Francesca Palumbo
fpalumbo@uniss.it

[1] DIEE, Universitá degli Studi di Cagliari, Cagliari, Italy

[2] CEI, Universidad Politecnica de Madrid, Madrid, Spain

[3] Dipartimento di Chimica e Farmacia, Universitá degli Studi di Sassari, Sassari, Italy

[4] Aitek S.p.A., Genoa, Italy

run-time approaches for image and video pipelines in CPSs. In many different domains image processing is a key aspect of a CPS. Indeed, in application domains such as video surveillance or environmental exploration, visual context and awareness are fundamental to making correct decisions and performing appropriate actions. As a test case scenario for this work, we consider a Convolutional Neural Network (CNN) for the classification of humans and animals. The reference network has been provided by one of the FitOptiVis Use Case providers, which built it for a critical infrastructure surveillance scenario. To be effective in the addressed scenario, such CNN needs to be efficient, with optimal execution time and power consumption, accessible to the user, both in terms of costs and usability, and adaptive, to trade the mentioned metrics according to the specific needs (e.g., lowering power consumption and, in turn, execution time when the reactivity of classification is not crucial). The best candidate target architecture for such a job is an FPGA based HW accelerator since it can deliver efficiency and adaptivity with limited costs. Regarding usability, however, such target architecture still constitutes a challenge for developers, especially if adaptivity is required. In fact, deriving an efficient accelerator for a given CNN model is hard, and several attempts already exist in literature in this sense, but making it adaptive at runtime is a feature that is not usually included in the available design flows.

To overcome these limitations, we assembled and developed a complete toolchain to derive HW accelerators for neural networks, where users can easily shape adaptivity yet at the model level. This flow is meant to enable the automatic definition of application-specific HW accelerators starting from the high-level Open Neural Network eXchange (ONNX) descriptions of the network to be executed. ONNX is a widely-adopted open format built to represent machine learning models. The peculiarity of these accelerators is the adaptivity support: by exploiting coarse-grained HW reconfiguration [4–6], the flow presented hereafter guarantees to the system the possibility of providing functional or non-functional reconfiguration. Such adaptivity can be modeled directly by the user at a high level of abstraction, so that it can be shaped according to the specific needs, requirements, and constraints. As a proof of concept, in this work we show how it is possible to provide different working points for a given CNN, trading off power for latency by changing resources dedicated to the processing of different CNN layers. These working points are only an example demonstrating the possibility of achieving adaptivity at the edge, which can be exploited for many different purposes, e.g. to switch among configurations providing different Quality of Service (QoS) versus energy consumption profiles [7], or to provide different encryption degrees at the cost of a higher power consumption [8], or to deliver different performance per power [9] or energy [10, 11] trade-offs. At the edge, minimizing consumption is of paramount importance and demonstrated to be challenging [12], which motivated in this work the choice of exploring different working points presenting different power profiles. The main advantage of the proposed flow is that the process of creating the adaptive accelerator is almost automated, from ONNX models to the ready-to-use accelerator: the user has only to specify working points at the same model level. Note that, even if the presented flow can be easily adapted to an Application Specific Integrated Circuit (ASIC) target, in this work we address only FPGA devices, which are surely advantageous in cost and development time with respect to the former.

The main contribution of the work is then an almost automated flow for developing neural network adaptive accelerators on FPGAs, from ONNX network specifications down to HW descriptions ready for logic synthesis. Such a flow allows users to shape adaptivity by acting on the model, thus avoiding unnecessary implementation details which can harden the design process. The same runtime adaptivity support makes it novel among the literature works providing neural network and CNN acceleration support. A proof of concept implementation of an adaptive CNN enabling different working points, each delivering a different execution time versus power trade-off, demonstrates the effectiveness of the proposed approach. In particular, the obtained implementation can lower the power consumed by each layer by 30% in front of an overall 8% overhead in terms of network classification time.

The rest of the paper is organized as follows: Section 2 gives an overview of the background, discussing aspects related to CNN acceleration, as available tool support and adaptivity for them; Section 3 describes the assembled flow from ONNX to adaptive accelerators, clearly showing the required user manual interventions, and the customization possibilities. Section 4 illustrates the preliminary results regarding the single layers and the overall accelerator, obtained with the proposed flow on the CNN adopted as an example for the whole explanation. Lastly, Section 5 provides some final remarks and future directions of the work.

2 Background

In this section the background of the work is presented, touching topics related to CNN acceleration and tooling (Section 2.1) and to tooling for generic HW acceleration and adaptivity (Section 2.2). The tools involved in the proposed toolchain are also presented.

2.1 CNN Acceleration and Tooling

A huge number of approaches have been proposed in literature focusing on CNN acceleration. Lots of them, those more oriented to adaptivity and reconfigurability, rely on FPGA architectures as a technology substrate [13]. Probably the most ready-to-use and powerful currently available FPGA-based acceleration engine is the proprietary one offered by Xilinx, which provides an integrated framework, called VitisAI [14], that helps designers in mapping CNNs on a template soft IP called Deep Learning Processing Unit (DPU) [15].

2.1.1 Architectures for CNN Inference

Most tool-oriented works focus on the creation of an optimized FPGA structure, usually relying on a reference architecture template, based on a specified target CNN. Yu et al. [16] developed an FPGA acceleration platform that leverages on a unified framework architecture for general purpose CNN inference acceleration at a data center, achieving a throughput comparable with the state-of-the-art Graphics Processing Units (GPUs) in this field, with less latency. Zhang et al. [17] proposed Caffeine, a HW/SW library to efficiently accelerate CNNs on FPGAs, leveraging on a uniformed convolutional matrix multiplication representation. Ma et al. [18] presented a Register Transfer Level (RTL) CNN compiler that generates automatically customized FPGA HW for the inference tasks of CNNs from SW to FPGA. These frameworks provide huge performance gains when compared to state-of-the-art accelerators, as well as to general purpose Central Processing Units (CPUs) and GPUs. However, they do not take into account adaptivity as a main development objective.

2.1.2 Resource-constrained and Real-time Solutions

Other approaches focus on resource-constrained deployment, mostly validated on smaller devices, e.g. Zynq System on Chips (SoCs), usually Z-7045 or smaller, or Zynq Ultrascale+ Multi-Processor System on Chips (MPSoCs). Venieris et al. [19] presented a latency-driven design methodology for mapping CNNs on FPGAs. As opposed to previously presented approaches, mainly intended for bandwidth-driven applications, this work targets real-time applications, relying on Xilinx High Level Synthesis (HLS) tools (i.e. Vivado HLS) for mapping, demonstrated on a relatively simple CNN such as AlexNet, and a very regular one such as VGG16 featuring only 3x3 kernels, providing a peak performance of 123 GOps on Xilinx Zynq Z-7045 SoC. Other works focus on a template-based approach relying on programmable or customizable RTL accelerators [18, 20, 21], more similar to the one that is used in this paper. SnowFlake [20] exploits a hierarchical design composed of multiple compute clusters. Each cluster is composed of four vector compute units including vector Multiply and Accumulate (MAC), vector max, maps buffer, weights buffers, and trace decoders. SnowFlake provides a computational efficiency of 91%, and an operating frequency of 250 MHz (best-in-class for CNN accelerators on Xilinx Zynq Z-7045 SoC). NEURAghe is an inference accelerator exploiting a HW convolution engine on FPGA [22]. The main computational engine of the accelerator is a matrix of MAC modules that takes care of the convolution workload. The accelerator is configurable at design time with different parameters. On a Xilinx Zynq UltraScale+ MPSoC ZU3EG device, it is possible to implement a configuration featuring a matrix of 90 MAC modules, distributed over 9 parallel input channels and 10 parallel output channels, working at 180 MHz clock frequency. NEURAghe has been implemented with flexibility in mind, which has also enabled its validation of time series analysis with Time Convolution Networks [23]. However, none of these architecture templates supports runtime adaptivity.

2.1.3 Reduced Data-precision Implementations

Multiple approaches have been focusing on performance improvement through the reduction of the precision of arithmetic operands. Most of the architectures use a precision of 16-bit (fixed-point) [18–20]. However numerous reduced-precision implementations have been proposed recently, relying on 8-bit, and 4-bit accuracy for both maps and weights, exploiting the resiliency of CNNs to quantization and approximation [21]. Even extreme approaches to quantization have been proposed, exploiting ternary [24] or binary [25] neural network accelerators for FPGA. This approach significantly improves the computational efficiency of FPGA accelerators, allowing them to achieve performance levels as big as 8 TOPS [24]. A recent work by Rasoulinezhad et al. [26], starting from the Xilinx DSP slices, proposed an optimized DSP block called PIR-DSP to efficient map 9, 4, and 2 bits data precision MAC operations. It is implemented as a parameterized module generator targeting both FPGAs and ASICs, reaching an estimated runtime energy decrease of up to 31% for a MobileNet-v2 implementation compared with a standard DSP mode. Other works, like Wang et al. [27], leverage on FPGA LUT blocks as inference operators for Binary Neural Networks achieving up to twice area efficiency compared to state-of-the-art binarized neural network implementation and against several standard networks models.

To the best of our knowledge, however, literature is still lacking an analysis of the possibility of adapting the design to multiple operating modes at run-time, when using FPGA-based acceleration for AI-related workload. In this work, we propose a toolchain that is suitable to derive run-time adaptive accelerators or accelerator components. Our approach is thus complementary to the previously mentioned pieces of work and can serve as an additional development layer in

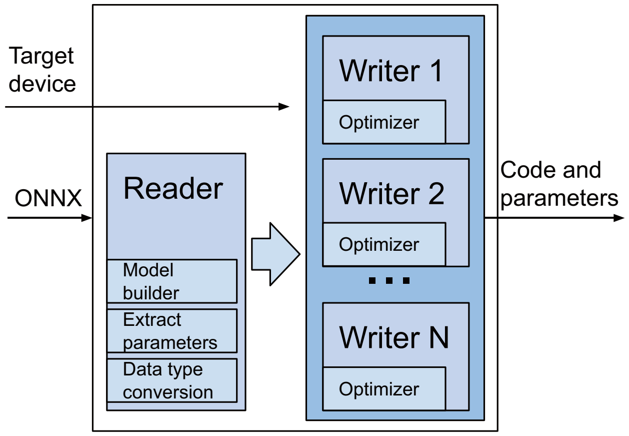


Figure 1 Internal structure of the ONNXparser.

use cases requiring fast and effective adaptation to varying scenarios.

2.1.4 ONNXparser

Dealing with CNN acceleration and related tooling, an interesting instrument, which is involved in the proposed flow, is ONNXparser[1]. This parser has been made available open source as a result of the H2020 ALOHA project [28]. It is a Python application intended to parse the ONNX models and automatically create the code for a given kind of target device. ONNXparser has a modular design that can be extended by the user adding support for additional code generation targets.

As shown in Fig. 1, the input point of the tool is the ONNX reader. It is in charge of parsing the ONNX and identifying an object for each operator and connection found in the file. The input file is read using the `onnx` python package. In case the ONNX file also includes the parameters of the model, these are exported in external files during the parsing. The reader creates an intermediate format with a list of objects describing actors and connections that are found in the ONNX under analysis. Each structure is populated with actor/edge parameters.

At this point, the tool uses a set of *writers*. Each *writer* can take the intermediate format as input and generate platform-dependent code. Additional *writers* can be created and customized for different platforms. The code generation also implements required optimization/transformation steps such as, for example:

- batch normalization folding: in some cases, by taking adequate measures, multiplications and additions required for the implementation of this actor can be integrated into weights and bias values for the convolution that follows the batch normalization in the CNN topology;
- data type conversion: the tool can perform an adaptation of floating-point represented value parameters into fixed point formats;
- operators merging: when accelerator architectures perform operations as a sequence in one single accelerator activation, the parser can merge the original operation, expressed in the ONNX in one single call of the composed primitive to the accelerator;
- data marshaling: different processing elements require different orderings of input data and weights, for example different feature/dimension interleaving. The *writer* in the parser can be easily instrumented to generate the code according to such an ordering.

ONNX parser currently supports most operators used in state-of-the-art CNNs and has been tested successfully on VGG, ResNet, YOLO, and UNet. *Writers* are available for different targets: plain C (the writer used in this paper), Pytorch, CMSIS-based CNN implementation, accelerators [29].

2.2 Design Automation for Hardware Acceleration and Reconfiguration

Design automation for digital systems is still an open issue. The main players on the market are recently pushing for an additional level of automation, besides the well-known logic synthesis and implementation. While this latter allows designing, and programming in case of FPGAs, digital systems by means of RTL descriptions in Hardware Description Languages (HDLs), newer techniques known as HLS raise the abstraction level of such descriptions.

2.2.1 High-level Synthesis

Most HLS tools use common programming languages for general-purpose systems, mostly C code, as input specification of the desired functionality. Lots of HLS solutions from academia and industry have been proposed in the last years [30], proof of the fact that the scientific community and the market are moving towards this new automation level. In particular, the main ASIC and FPGA tool vendors all deliver HLS solutions within their design suites [31, 32]. Despite being promising and constantly under development, current HLS instruments are still not capable of filling the gap between users and technology. In fact, to be effective for generating optimized usable HW, HLS requires to act on the source code with refactoring and pragmas insertion. Efforts and skills for such pre-processing phase, usually made manually by the same user, could overcome the overall automation benefits. Moreover, the maturity of HLS tools

[1] https://gitlab.com/aloha.eu/onnxparser

at the moment is not enough for tackling aspects that may result crucial in different contexts, such as power consumption, system integration or reconfiguration [33].

2.2.2 Adaptivity Support

Reconfiguration is commonly used in specialized logic to reach adaptivity. Designing and optimizing reconfigurable digital systems has historically been a challenge for developers [34]. Such a challenge is even harder for modern CPSs which may have to cope with strict and multiple constraints and requirements, such as extremely low power behavior or real-time reactivity. For instance, when reactivity is crucial, coarse-grain reconfiguration could be more suitable than a fine-grain (e.g. Dynamic Partial Reconfiguration (DPR) on FPGAs) solution [5]. Indeed, performing reconfiguration at the entire data word level, like for coarse-grain reconfigurable solutions, ensures quick reconfiguration time at the price of lower flexibility with respect to a fine-grain solution, where reconfiguration is applied at the single-bit level. Reactivity and, in turn, reconfiguration, as well as other secondarily but often not less important aspects in CPSs design, are hardly addressed by the currently available HLS instruments. The main reason behind the existing lack is that to be effective, solutions tackling reactivity, power, and other secondary and non-functional aspects require a robust model-based approach [35, 36].

2.2.3 Model-based Design: Dataflow Advantages

Model-based design already proved to be successful in the abstraction of low-level details, enabling quick design automation. Dealing with HW acceleration, dataflow models turn out to be particularly suitable in this sense, including when adaptivity is necessary. Dataflows can be seen as directed graphs whose nodes, namely actors, represent processing elements, while arcs represent point-to-point communication channels between actors. Such channels hold storing resources and chunks of data flowing throughout them, namely tokens, are managed according to a First-In First-Out (FIFO) policy. Actually, HLS tools are indeed model-based, but according to the adopted input specification, the considered models can have different levels of abstraction. Here, common programming languages, such as C, can be seen as models of the desired system (e.g., they abstract time information), but they are not very suitable for highlighting modularity and parallelism of this latter, features of primary importance in digital systems design. Dataflows are instead intrinsically representing parallelism of applications: for this reason, they have also been proposed as input specifications for HLS [37, 38]. However, the main drawback of such approaches is that users have to learn new languages and specifications, which can be far from the code they usually deal with. Here, a combination of dataflow models, for describing the system-level view of the desired platform, and common programming languages, for describing processing elements of the same system, can be the right trade-off between effectiveness and usability.

Modularity is very important in digital systems design when reconfiguration is required. Having a modular view of the application allows the identification of common resources to be shared among different configurations. For this reason, dataflows have already been employed as models for applications targeting either customizable substrates of generic processing units (executing actors) and FIFOs [4], as well as specialized reconfigurable computing accelerators [39]. In particular, this last case offers enough room for the previously mentioned mixed HLS model-based approach, employing both dataflows and common programming languages, taking the best of both.

2.2.4 MultiDataflow Composer Tool

A tool that provides runtime coarse-grained reconfigurability, leveraging on a dataflow description of the application, is proposed in [39]. The Multi-Dataflow Composer (MDC) is an open source tool for, optionally reconfigurable, HW acceleration support[2]. It takes as input applications that need to be accelerated, specified as dataflows: they can be different applications with common processing steps (common actors), to obtain functional reconfiguration or different versions of the same application, to obtain non-functional reconfiguration. The dataflows corresponding to such applications are combined together, and the resulting HDL top module is automatically generated (see Fig. 2). The HDL of the actors is not generated by MDC and has to be provided by the user (HDL component library): a common programming language can be adopted to model the actor behavior and to derive the corresponding HDL through HLS. Besides the baseline feature of combining applications and delivering a reconfigurable top module of the resulting HW accelerator, thanks to dataflows MDC is also capable of providing advanced features, such as optimizations under different design goals (resources, power, frequency), smart power management, and easy system integration.

In this work, to provide a full design flow for CNN acceleration and adaptivity support, MDC and a well-known HLS tool, Vivado HLS, have been combined with specific CNN instruments, ONNXparser, delivering a new powerful mean for the final users.

[2] https://github.com/mdc-suite/mdc

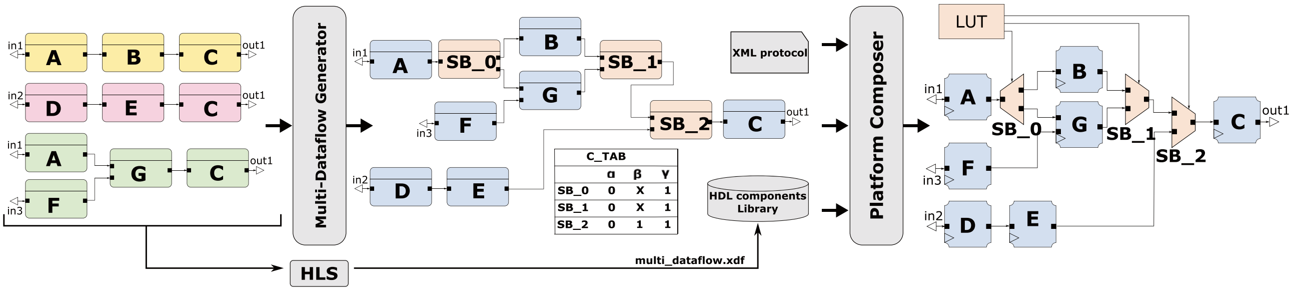


Figure 2 Overview of the MDC functionality, inputs and outputs. The three input dataflows (α, β, γ) are merged by the Multi-Dataflow Generator in a unique dataflow that shares common actors (A, C) and addresses tokens depending on the configuration using switching boxes (SB). Platform-Composer maps this dataflow in a hardware accelerator using the provided actor library (*HDL components library*) and communication protocol (*XML protocol*). The actor library can be handwritten or synthesized using any HLS tool.

3 Proposed Flow

In this section, we are going to describe our proposed flow to develop adaptive HW neural network accelerators adopting a dataflow-based approach. Several tools, the ONNXparser, Vivado HLS, and the MDC, are adopted and very limited user intervention is required. In particular, the starting CNN models fed to the ONNXparser will be described in Section 3.1. Then, input preparation for the following adopted tools, Vivado HLS and MDC, starting from ONNXparser output, are detailed in Section 3.2. Lastly, MDC potentials in delivering reconfiguration, and thus adaptivity, are shown in Section 3.3.

An overview of the proposed flow is depicted in Fig. 3, where labels are defined into red (dataflow networks generation) and blue (dataflow actors generation) circles to better understand the flow in the following detailed explanation. The flow is almost automatic. As shown in Table 1, there are two steps that are manual at the moment: *N1*, which is the ONNX to dataflow (input model for MDC) translation, and *A2*, which is the pragmas insertion and code refactoring on the C code generated by the ONNXparser. In particular, the latter is intentionally manual since it constitutes, together with the input ONNX modeling, a point where users can shape adaptation.

3.1 CNN Modelling - ONNX Input

The CNN taken as proof of concept for the proposed work is based on InceptionNet v2 structure presented in [40], which constituted an important milestone in the development of CNN classifiers. Differently from previous fully-connected approaches, seeking to improve performance by stacking convolutional layers deeper and deeper at the price of significantly high elaboration cost (making them computationally inefficient), InceptionNet introduced a sparsely connected architecture. The sparse architecture is composed of modules performing multiple convolutions with multiple filter sizes in parallel, before concatenating the output and sending it to the next module. In this manner, the performance of the network can be optimized by balancing the number of filters per stage (width) and the number of layers (depth) in the network that, in turn, allows for reducing the number of parameters, as well as the amount of required memory and computation power. Moreover, having several filters with different sizes allows to better classify image information. In fact, in general, there is a huge variation in the location of the information inside an image, so choosing once and for all the right kernel size for convolution operation might be difficult. InceptionNet allows for subsequent refinements: the scale is small and local at first, and then it gets bigger as it goes deeper.

Table 1 Brief description of each step of the flow, highlighting the utilized tool and its input and output.

Step	Tool	Input	Output	Description
A1	ONNXparser	ONNX model	C model	Converts the ONNX model to an executable C model
A2	Manual	C model	Synthesizable C model	Refines the C code of actors to shape adaptivity
A3	Vivado HLS	Synthesizable C model	RTL model	Synthesizes C actors to generate the actors library
N1	Manual	ONNX model	XML dataflow models	Converts the ONNX model to a dataflow to shape adaptivity
N2	MDC	XML dataflow models	Merged XML dataflow model	Merges multiple dataflow models in a reconfigurable datapath
N3	MDC	Merged XML dataflow model + RTL model	RTL reconfigurable accelerator	Generates the final hardware accelerator

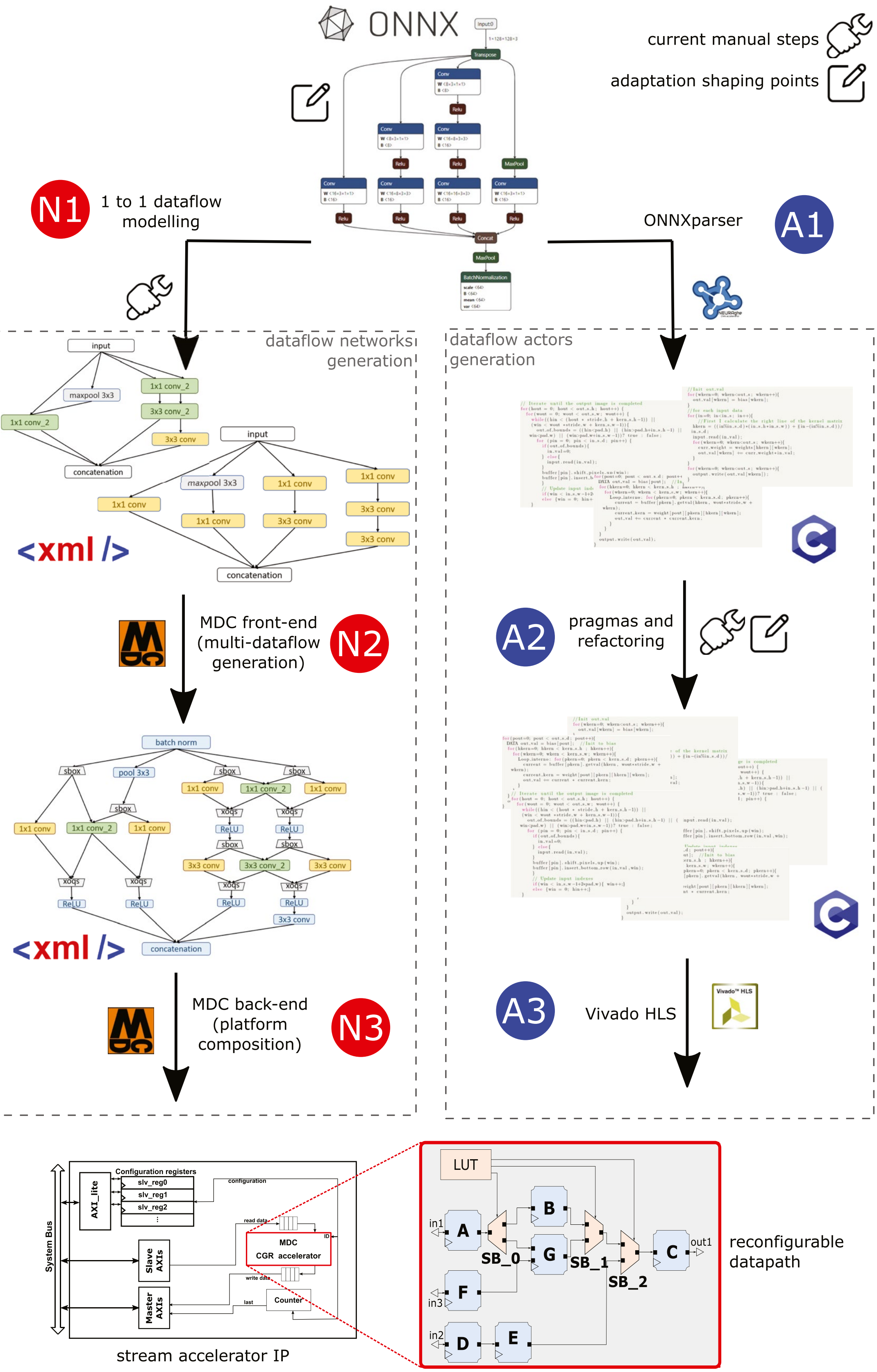


Figure 3 Proposed automated design flow for CNN adaptive HW accelerators. Red and blue circles define labels for the different steps which are referenced in the related explanation.

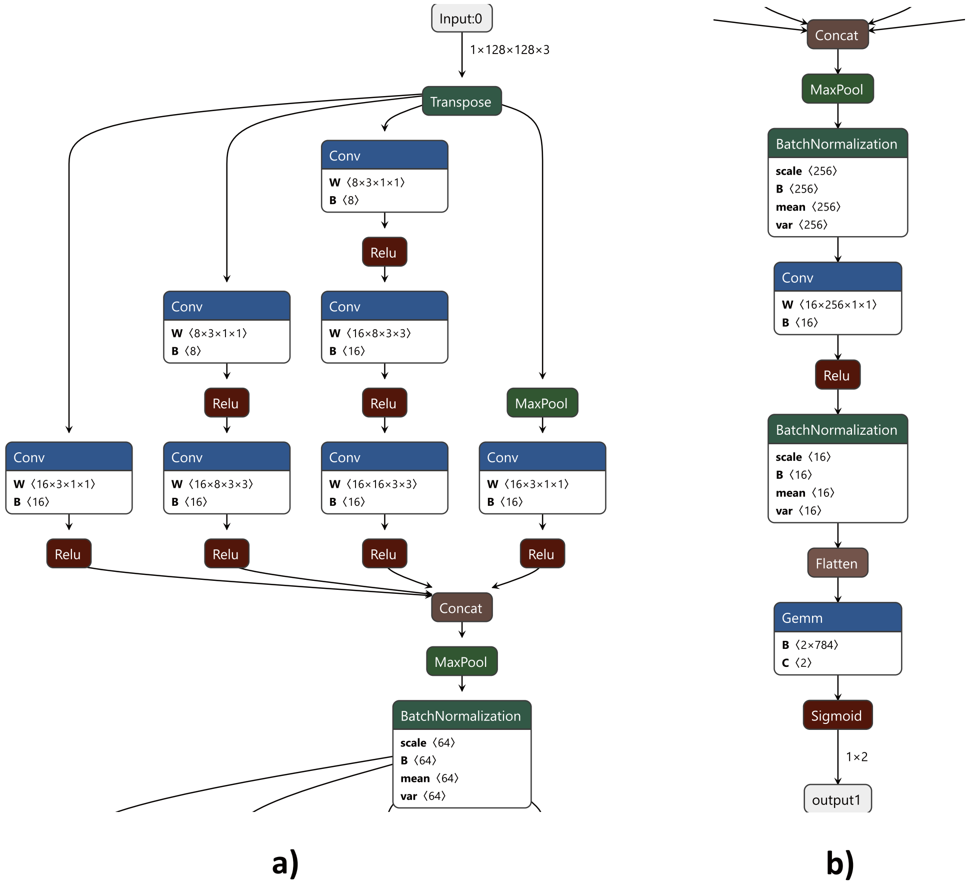


Figure 4 InceptionNet structure details: **a** inception module with the four parallel branches; **b** final stage.

Adopted Network In this paper, as already said in Section 1, we adopted a network classifying humans and animals for the surveillance of a water supply infrastructure, within the context of the FitOptiVis project use case 1 [41] The composition of the InceptionNet module is based on InceptionNet v2, in which each module contains four parallel branches whose outputs are concatenated at the end (Fig. 4a):

- a 1x1 convolutional layer (Conv);
- a 3x3 max pool layer (MaxPool) followed by a 1x1 Conv layer;
- a 1x1 Conv layer followed by a 3x3 Conv layer;
- a 1x1 Conv layer followed by two 3x3 Conv layers.

The overall structure of the adopted CNN was modified compared to the original network to make it compatible with ONNXparser, as described in Appendix 1. Input data are provided as tensor with format NCHW (batch size, channel, height, width), characterized as [1, 3, 128, 128] in this case, and with data type *float32*. This input is sent to 3 cascaded modules with the above-mentioned structure. After each convolution operation, the activation function ReLu is applied [42]. The outputs of the four branches are concatenated and provided to a MaxPool layer, using a 2x2, 3x3, and 3x3 kernel for each cascaded module respectively, and then normalized through a batch normalization layer (BatchNormalization), with parameters *epsilon* and *momentum* defined by default in [43]. After the three cascaded convolutional modules, the output is forwarded to a new 1x1 Conv layer with ReLu activation and normalized again through a BatchNormalization layer (Fig. 4b). The returned tensor is flattened, and prediction is calculated using the Sigmoid [44] activation function, which gives a result in the range [0, 1].

In order to exploit the ONNXparser flow (step *A1* in Fig. 3), the CNN has to be provided in ONNX format. In this case, it has been modeled with Pytorch and then converted to ONNX. Modeling it in Keras is, in theory, also possible but several conversion issues are present at the moment (see detailed discussion in Appendix 1).

Training and Validation In response to the classification problem that the CNN wants to solve, two types of images are required: humans and animals. The training has involved the UTKface [45], which is a large-scale face dataset[3] with

[3] Each image is annotated with different tags (age, gender, and ethnicity), and a large variation in pose, facial expression, illumination, occlusion, and resolution is covered.

over 20 thousand face images, and the Animals-10 [46], which contains about 28 thousand medium quality animal images[4]. Five thousand images have been chosen from each of the considered datasets (about 25% of the UTKface dataset and about 18% of the Animals-10 one), being the variety the reason for their choice, for an overall number of 10 thousand images. In this way, it is possible to get smaller, but still representative datasets. The training has been performed in 10 epochs with 100 steps each.

With the rest of the non-used images from UTKface and Animal-10 datasets, four testing sets have been defined. These sets have a size corresponding to 5% (*val_data_1*, 500 images), 10% (*val_data_2*, 1000 images), 25% (*val_data_3*, 2500 images) and 50% (*val_data_4*, 5000 images) of the training dataset, and contain different images from those belonging to the training set.

Images have been treated before their usage in the training, to meet the expected size:

- The images coming from the considered datasets, initially with shape format HWC and data in the range [0, 255], have been resized to fit in network input data size 128x128, and they have been converted to a `torch.FloatTensor` of shape CHW in the range [0.0, 1.0].
- Additionally, data have been normalized using the using `torchvision.transforms.Normalize` function. This transform normalizes each channel of the input tensor taking into account the mean and standard deviation of each channel.

In order to measure the results obtained during the training phase, parameters such as accuracy and training losses have been monitored. Table 2 depicts the final validation results.

Table 2 Training and Validation results on the different considered validation sets.

Training				
Batch size			32	
Epochs			10	
Steps per epoch			100	
Average accuracy			98%	
Average loss			3.2	
Validation				
val_data	*_1*	*_2*	*_3*	*_4*
accuracy	96%	96%	98%	98%

3.2 Deriving Inputs for HLS and MDC

The starting point of the proposed flow is an ONNX representation of the CNN, resulting from the preliminary modeling discussed in Section 3.1. Vivado HLS allows us to easily customize data representation formats. We used a 16-bit fixed-point representation as this is a common choice on FPGA implementation to efficiently utilize the programmable logic (Section 2.1.3).

N1 - Mapping to Dataflow Each layer of the ONNX model is mapped into one, or more, dataflow actors. This step (*N1* in Fig. 3) is straightforward since in an ONNX model a node represents one layer, and in a dataflow graph, a node represents an actor. Moving to the arcs, in an ONNX model they represent a tensor, while in a dataflow model, an arc represents a FIFO channel in which a stream of data can flow. To convert a tensor into a stream of data it is necessary to choose a streaming order for the elements of the tensor. We opted for the raster scan order since it is consistent with the structure ordinarily used to perform the convolution. We will see afterward that the adoption of raster scan relieves the system of buffering the whole tensor between two actors.

A1, A2, A3 - Synthesis of Actors The next step is to build an HDL description of the actors (steps *A1*, *A2* and *A3* in Fig. 3). To do so, we used the ONNXparser and Vivado HLS tool, introduced respectively in Sections 2.1.4 and 2.2.1. The first is able to convert an ONNX model to the corresponding C description (step *A1*), where each layer is implemented through a different C function. The second is a HLS tool that generates the HDL code implementing a C function given as input (step *A3*). Nevertheless, some modifications have been necessary to make the C code coming from ONNXparser synthesizable by Vivado HLS, and then to optimize it (step *A2*):

- Inserting a line buffer actor to store part of the input stream;
- Inserting directives to optimize execution;
- Adding actors to store and share weights;

The following section describes these modifications, which correspond to one of the two available points to the user for shaping adaptivity (the other is *N1*).

A2 - Convolutional Layer A great deal of effort went into building the convolutional layer. As said, it is not possible for an actor to have random access to the input tensor, but the input is received as a stream of data. Given that, a certain storage capacity is needed within a convolutional layer to

[4] The images in this dataset are taken from google images, and belong to ten different categories, among which are mammals, reptiles, insects, birds, and fishes.

```
for(hout = 0; hout < out_s_h; hout++) {
 for(wout = 0; wout < out_s_w; wout++) {
Read Loop:
  while( (win <= (wout * stride_w + kern_s_w
    -1)) ||
     (hin < (hout * stride_h + kern_s_h-1) )
    ){
   out_of_bounds = ((hin<pad_h) || (hin>
    pad_h+in_s_h-1) ||
      (win<pad_w) || (win>pad_w+in_s_w-1)) ?
      true : false;
   for (pin = 0; pin < in_s_d; pin++) {
    if(out_of_bounds){in_val=0;}
    else{ input.read(in_val);}
     buffer[pin].shift_pixels_up(win);
     buffer[pin].insert_bottom_row(in_val,
    win);
     }
     if(win < in_s_w-1+2*pad_w){ win++;}
     else {win = 0; hin++;
     if(hin > (hout * stride_h + kern_s_h-1)
    ){break;}
  }}
Write Loop:
   for(hkern=0; hkern < kern_s_h ; hkern++){
    for(wkern=0; wkern < kern_s_w; wkern++){
     for(pkern=0; pkern < kern_s_d; pkern++)
    {
      out_val = buffer[pkern].getval(hkern,
    wout*stride_w+wkern);
      output.write(out_val);
}}}}}
```

Listing 1 C code of the line buffer actor. The first two for loops are responsible for iterating over the output tensor to check whether it has been completed or not: for each pixel of the output the Read Loop and the Write Loop must be executed. The Read Loop reads from the input stream until a submatrix coherent with the kernel size has been received. At this point, it exits letting the Write Loop forward the data to the convolutional actor in the proper order

store part of the stream. The reason lies in the convolutional operation itself, which uses the same element of the input tensor multiple times to compute different elements of the output. For example, in a layer with a 3x3 kernel and stride 1, each element is used nine times. So, we implemented a convolutional layer with two actors: a line buffer actor and a convolutional actor. The line buffer is responsible for receiving data, storing them, and forwarding them to the convolutional actor (see Listing 1). More in detail, it stores elements row by row until a sub-matrix, consistent with the kernel size, is stored. At this point, it is possible to calculate an element of the output, so the line buffer actor forwards this sub-matrix to the convolutional actor to perform the actual convolution. To do that, the line buffer must have the capacity of storing a number of rows equal to the height of the kernel.

A2 - Optimizing Through Directives After the ONNX-parser elaboration (*A1* in Fig. 3), a library of C functions corresponding to dataflow actors would be ready to be processed by Vivado HLS, but a further step of optimization is needed to exploit the capabilities of the tool. As before, we concentrated the most of our effort on the convolutional actor, being it the heaviest in terms of computation and, consequently, potentially the one promising the larger possible improvements on the overall performance of the system. In particular, the convolution function is described through a set of nested loops. The innermost loop consists of three operations: reading an element from the kernel, multiplying it with the last input element, and accumulating it (see Listing 2). These three operations must be performed one after the other. But there is no dependency between consecutive iterations of the loop, thus an iteration can start even before the previous one is completed, leaving the door open for pipelining. This opportunity to speed up the execution is hidden by the imperative C description, the synthesizer can be forced to apply pipelining during synthesis with the `pragma HLS PIPELINE` directive, as depicted in the same Listing 2.

```
for(hout=0; hout<out_s_h; hout++){
 for(wout=0; wout<out_s_w; wout++){
Init loop:
   for(init_idx=0; init_idx < out_s_d;
     init_idx++){
    bias.read(current_bias);
    out_val[init_idx] = (DATA) current_bias;
   }
Convolution loop:
   for(hkern=0; hkern < kern_s_h ; hkern++){
    for(wkern=0; wkern < kern_s_w; wkern++){
     for(pkern=0; pkern < kern_s_d; pkern++){
      input.read(current);
Inner loop:
      for(pout=0; pout < out_s_d; pout++){
      #pragma HLS PIPELINE rewind
       weight.read(current_kern); //
     current_kern = weight[pout][pkern][hkern
     ][wkern];
       out_val[pout] += current *
     current_kern;
   }}}}
Write loop:
    for(wr_idx=0; wr_idx < out_s_d; wr_idx++)
     {
     output.write(out_val[wr_idx]);
}}}
```

Listing 2 C description of the convolutional actor. The first two for loops are responsible for iterating over the output tensor: for each pixel of the output, the vector out_val must be initialized to the bias (Init loop), convolved with the kernel (Convolution loop), and then written to the output channel. The pragma HLS PIPELINE is placed in the innermost loop to speed it up

A2 - Access to Weights In order to support reconfigurability, we need further refinement in the implementation of the convolutional layer. In fact, storing all the weights used by a CNN requires a considerable memory footprint. To avoid slow memory accesses to off-chip memories, we wanted to keep all the weights inside the FPGA chip. Moreover, in line with MDC philosophy which supports different application configurations by replicating and sharing actors/resources (see Section 2.2), we decided to opt for sharing the weights among configurations, rather than duplicating them for each configuration. To do so, weights that are in common between different configurations are packed within the same actors, and these actors have been inserted in the dataflow model of the different configurations.

Another desirable feature is to keep the implementation consistent with the dataflow paradigm, which means avoiding random access to weights and providing them in a streaming manner. To do that, each set of weights is mapped on a different, dedicated actor. This type of actor has only one output port and they are in charge of storing the kernel weights and delivering them in the proper order to be used by convolutional actors. The same choice has been taken for bias storing, so that dedicated bias actors have been modeled.

A2 - Resulting Convolutional Layer Wrapping up, the final result is that a convolutional layer is mapped in four different actors: a weight and a bias actor to store the kernel, a line buffer actor to store parts of the input stream to be reused, and a convolutional actor to perform the computation. The overall schematic of the layer, which in turn takes the name of baseline layer (*LAYER_B*), is depicted in Fig. 5. Note that *LAYER_B* includes the baseline version of the convolutional actor, called hereafter *CONV_B*, which elaborates one convolutional layer.

A2 - Other Layers We used the same approach of splitting a layer into two actors, one for storing incoming data and one for performing the computation, to implement the Max-Pool layers. Thus they are mapped into a line buffer actor, exactly as the one saw before, and a pooling actor. Instead, we used a single actor for mapping the other types of layers: normalization layer, ReLU layer, and concatenation layer. In these cases there is a direct correspondence between an input element and an output one, so no storing is needed. The same holds for the fully connected layers, where each input element is multiplied and accumulated a number of times equal to the output size. In a fully connected layer, each element of the output vector depends on the whole input, thus the partial sum for each output is stored inside the actor until the whole input is received. Moreover, in all these cases there is no need for additional dedicated actors for storing (and sharing between configurations, as will be more clear in Section 3.3) weight and bias data, which are not present in layers other than convolutional.

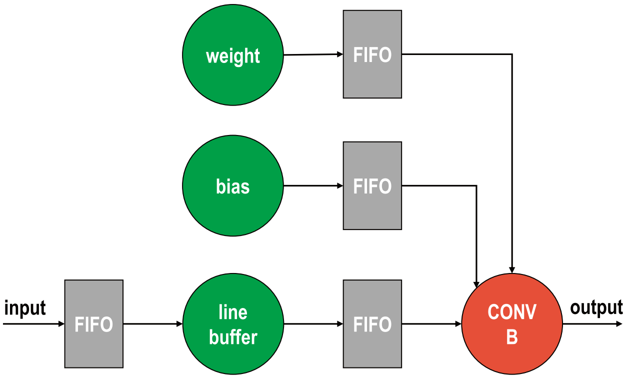


Figure 5 Schematic of the implementation of a convolutional layer. The weight and bias actors (top green circles) are sources that store and provide the parameters of the kernel. The line buffer actor (bottom green circle) stores part of the input stream and forwards it when it is possible to evaluate an output element. The *CONV_B* actor performs the convolution (red circle).

3.3 Adaptive Accelerator Generation

In the proposed flow, adaptivity is delivered by the MDC tool that generates a reconfigurable datapath, and more precisely a whole accelerator, capable of accelerating all the applications corresponding to a set of input dataflow models (step *N2* in Fig. 3). Here, reconfiguration can be functional, when the input dataflow models correspond to different applications, or non-functional, when they model different working points (e.g. with different speeds, quality, and power consumption) of the same application. Reconfiguration is performed at the dataflow actors level, meaning that MDC shares the common actors among different input dataflows through additional switching modules (sboxes), so that in the resulting system the corresponding resources are multiplexed in time among the executed configurations.

N1 - Exploiting Inception Topology For the considered CNN case, to achieve some degree of adaptivity we exploited the peculiar architecture of InceptionNet to map more than one layer on the same convolutional actor, thus sharing the underlying resources (DSP slices) and implementing non-functional reconfiguration. As seen in Section 3.1, InceptionNet is characterized by four branches working on the input in parallel, where one or more convolutional layers are executed. The regularity of the topology leads to the fact that some convolutional layers within the same block share the same hyper-parameters (e.g. kernel size, stride, zero-padding). This allows using the same actor, which has access to different weights, to perform different convolutions. We will refer to this actor as *CONV_D* actor, and we will let it be capable of performing two different convolutions (double

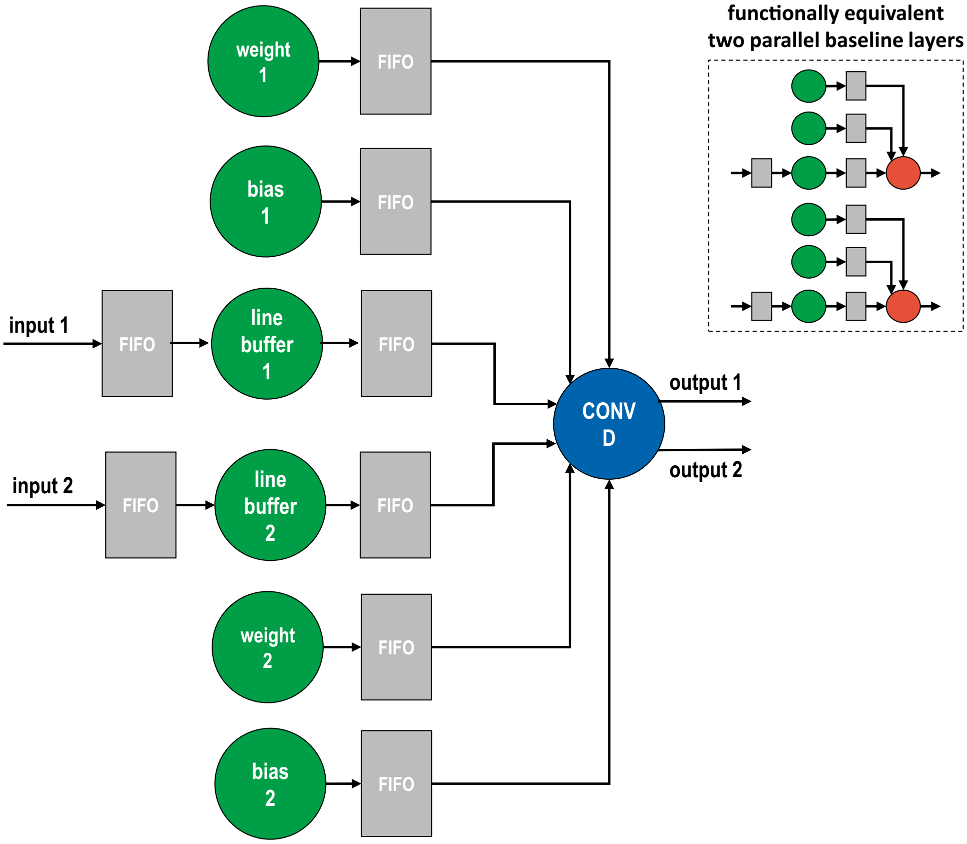


Figure 6 Schematic of the implementation of two convolutional layers using a single convolutional actor, which we will refer to as *CONV_D*. The *CONV_D* actor (blue circle) performs the convolution of two different inputs with two different kernels, but the two layers share the same hyper-parameters. On the top right corner, the functionally equivalent two parallel baseline layers (*LAYER_B*) are depicted.

convolutional actor). Then two convolutional layers can be mapped into three actors: two line buffer actors, one for each flow, plus a single *CONV_D* actor. In Fig. 6 the resulting implementation of the two merged layers (*LAYER_D*) is shown. Of course, besides convolutional (double) actors and line buffers, also weight and bias actors, one per each implemented layer, are included. Note that, this process involves both code refactoring and dataflow modeling activities (step *A2* and *N1* in Fig. 3 respectively). This implementation is expected to achieve adaptivity through non-functional reconfiguration: it should reduce resource utilization and, in turn, power consumption, while increasing the execution latency with respect to a fully parallel baseline implementation.

N2 - Reconfigurable Layer The possible implementations of two convolutional layers, *LAYER_B* which employs two parallel layers (Fig. 5 shows only one of them) and *LAYER_D* with double layers just introduced, share many actors. The line buffer actors, the weight actors, and the bias actors are in fact common to both of them. The *CONV_B* and *CONV_D* are the only different actors between *LAYER_B* and *LAYER_D*. By applying this simple modification to the baseline structure of the CNN dataflow model (which adopts only *LAYER_B* and *CONV_B*, and hereafter named *CNN_B*), a different working point (hereafter named *CNN_D*) has then been derived substituting different compatible couples of *LAYER_B* with the corresponding *LAYER_D*. Table 3 gives an overview of the overall dataflow models composition: *CNN_B* involves 131 actors, while *CNN_D* 121. Overall, the two models have 113 actors which can be shared.

Through the MDC tool it has then been possible to combine *CNN_B* and *CNN_D* to obtain the corresponding reconfigurable dataflow model (step *N2* in Fig. 3), hereafter named *CNN_R*, and the related datapath for the adaptive accelerator (step *N3* in Fig. 3). Please note that the HDL corresponding to the dataflow actors is given by Vivado HLS (step *A3* in Fig. 3) taking as input the C code defined and refined as discussed in Section 3.2 and at the beginning of this Section.

N3 - Accelerator Generation The accelerator implements the two versions of the network and switches dynamically from the execution of one of them to the other. FIFOs dimensions have been assigned empirically for all the considered dataflow networks. In particular, we sized the FIFOs of each branch (right before the concatenation actor) to avoid deadlocks. To do that we considered the different ratios between received input elements and output elements produced by each branch. This ratio, in turn, depends on the number of layers and their kernel size. The *CNN_R* overall involves 140 actors plus 72 sboxes. A scheme of a single reconfigurable convolutional layer, where *CONV_B* and *CONV_D* are multiplexed in time through sboxes and other actors are

Table 3 Dataflow models composition between baseline (CNN_B) and double (CNN_D) versions. Convolutional actors are the only ones differing from the two versions, the other types of actors/layers present in the model (Section 3.1) are shared.

Actor/layer type	# of instances in CNN_B	# of instances in CNN_D	# of shared instances
Batch normalization	4	4	4
Bias	22	22	22
Concatenation	3	3	3
Conv_B	22	4	4
Conv_D	0	9	0
Gemm	1	1	1
Line buffer	22	22	22
Line buffer MP	6	6	6
Maxpool	6	6	6
ReLU	22	22	22
Weight	22	22	22
TOTAL	130	121	112

shared, is depicted in Fig. 7. This layer is obtained, basically, by merging two parallel *LAYER_B* and the corresponding *LAYER_D*. The overall adaptive accelerator, as detailed in Section 4.1, embeds the reconfigurable datapath in a ready-to-use Xilinx IP provided with input/output stream channels, in particular AXI-Stream protocol compliant, which allows feeding the accelerator with tensors lying in main memory, accessible from a general purpose core, through a Direct Memory Access (DMA) engine.

4 Experimental Results

In this section, experimental results obtained with the proposed flow for the presented test case will be shown. Besides an overall view of the obtained CNN accelerator, coming from a real implementation and execution of the system on a development board (see Section 4.1), also a focus on single actor/layer data, coming from logic syntheses and HW simulations, is provided (see Section 4.2). In both cases, the target device is a Xilinx Zynq Ultrascale+ SoC, the XCZU9EG-FFVB1156, available on the ZCU102 Evaluation Kit, while the adopted tools are all from the Xilinx Vivado design suite.

4.1 Overall Accelerator

In this section, experimental results related to the overall CNN accelerator are reported. All the data refer to real implementations of the system on the considered ZCU102 development board. Besides the CNN datapath provided by MDC starting from CNN dataflow models (red square on the bottom right side of Fig. 3), an entire integrated processor-accelerator system is considered. As depicted in Fig. 8, the system is composed by:

- a host processor to control accelerator and data flowing (zynq_ultra_ps_e_0);
- a main memory where input tensor and results are stored (zynq_ultra_ps_e_0);
- a DMA engine to transfer data from/to the main memory to/from the accelerator performing memory mapped to stream protocol translation (axi_dma_0);
- AXI-Stream FIFOs to buffer accelerator inputs and outputs (axis_data_fifo_in_0 and axis_data_fifo_in_1);
- an AXI bus and an AXI smart connect to connect the processor, the DMA and the accelerator (ps8_0_axi_periph and axi_smc respectively);
- an adaptive CNN accelerator, which embeds the reconfigurable CNN datapath corresponding to the MDC input dataflow models (s_accelerator_0).

All the listed components, but the host processor and the memory which are hardcore, are instantiated on the Zynq device available programmable logic and run at 100 MHz.

Three different designs are considered for the experiment:

- *ACCEL_B*: the system provided with a non-adaptive accelerator embedding a CNN datapath coming from baseline *CNN_B* dataflow;
- *ACCEL_D*: the system provided with a non-adaptive accelerator embedding a CNN datapath coming from double *CNN_D* dataflow, which computes two convolutional layers with the same actor in different parts of the network as explained in Section 3.3;
- *ACCEL_R*: the system provided with an adaptive accelerator embedding reconfigurable *CNN_R* datapath obtained with MDC by combining *CNN_B* and *CNN_D* dataflows.

ACCEL_B and *ACCEL_D* designs are taken as terms of comparison for the adaptive *ACCEL_R* design.

Resource Utilization Figure 9 depicts the overall accelerator results of the considered systems. Here we can see that most of the resources in the programmable logic are used by the CNN accelerator in all the considered cases. Contributes from the DMA, the AXI FIFOs and AXI Buses are equal for each accelerator. These blocks are indeed part of the processor-accelerator interface and are required to communicate with the host in the integrated system, but do not depend on the executed accelerator itself. Moreover, the resource

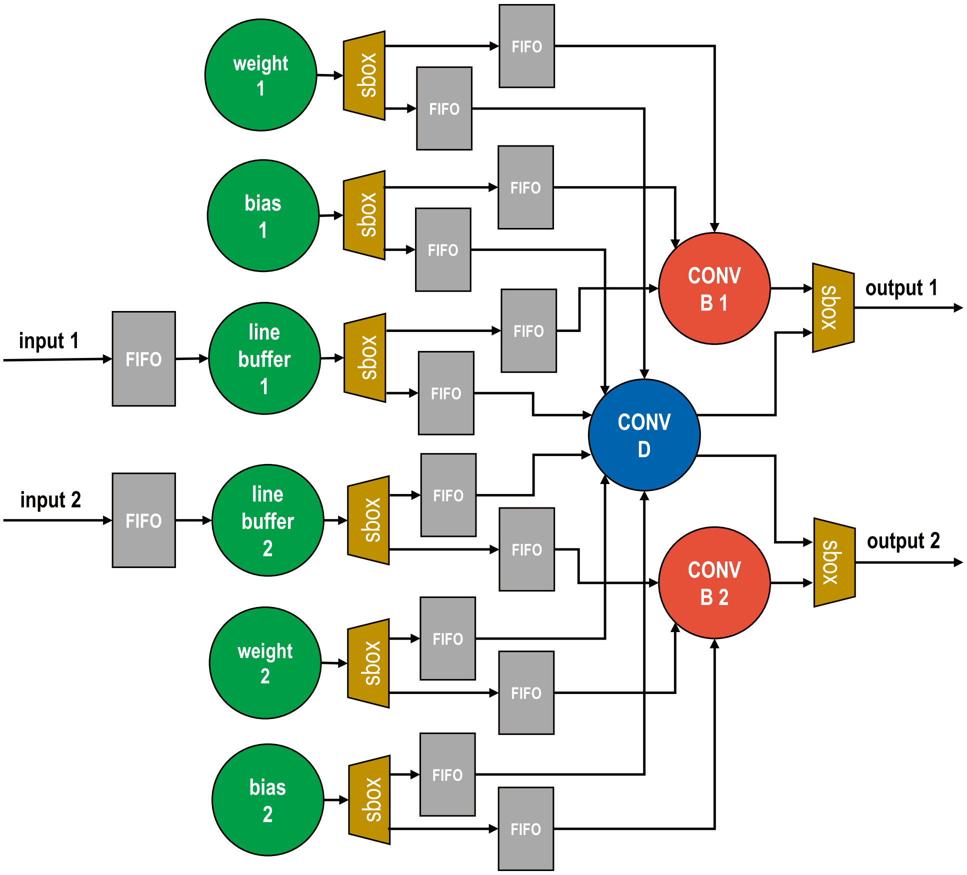


Figure 7 Schematic of the implementation of two convolutional layers using MDC. The green actors are shared between the two configurations, while the blue ones belong to one or the other. The sbox modules are responsible for directing the tokens according to the selected configuration.

overhead of the *ACCEL_R* is limited when compared with the two non-adaptive accelerators. This means that most of the resources are shared when the two accelerators are merged. From another point of view, the great deal of saving obtained sharing the common resources in *ACCEL_R* is appreciable looking at the comparison of CNN_R with the sum of two non-adaptive accelerators, CNN_B and CNN_D, shown in Fig. 9f. For every kind of resource, excluding DSPs, more than 40% of saving is present. This means that most of the resources, considering both actors and FIFOs, are shared. On the contrary, the kind of resource with the smallest saving is DSP, since DSPs are mainly used by the convolutional actors that are not shared between the two configurations. The overall result is that resource sharing allows the reconfigurable accelerator *ACCEL_R* to deliver adaptivity at a reduced resource cost compared with a system that implements two separate non-adaptive accelerators (*ACCEL_B* and *ACCEL_D*) and switches dynamically between them.

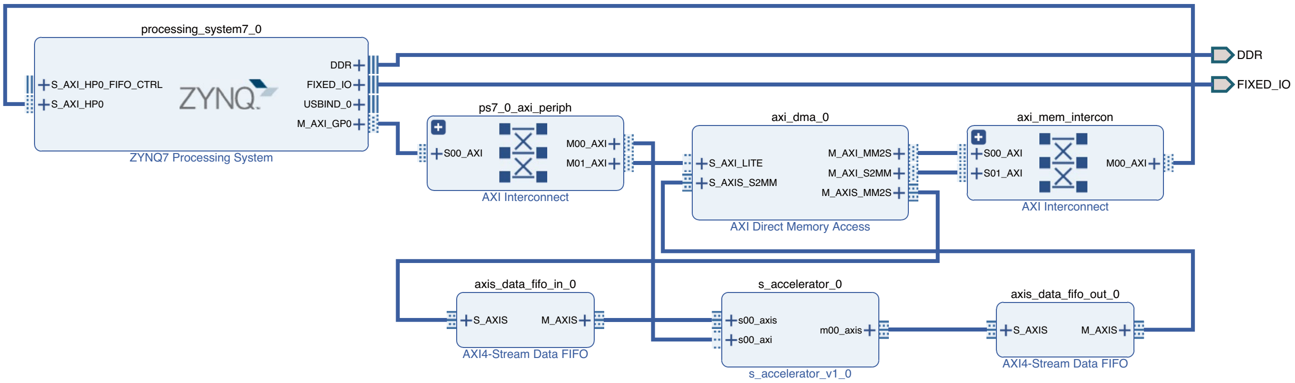
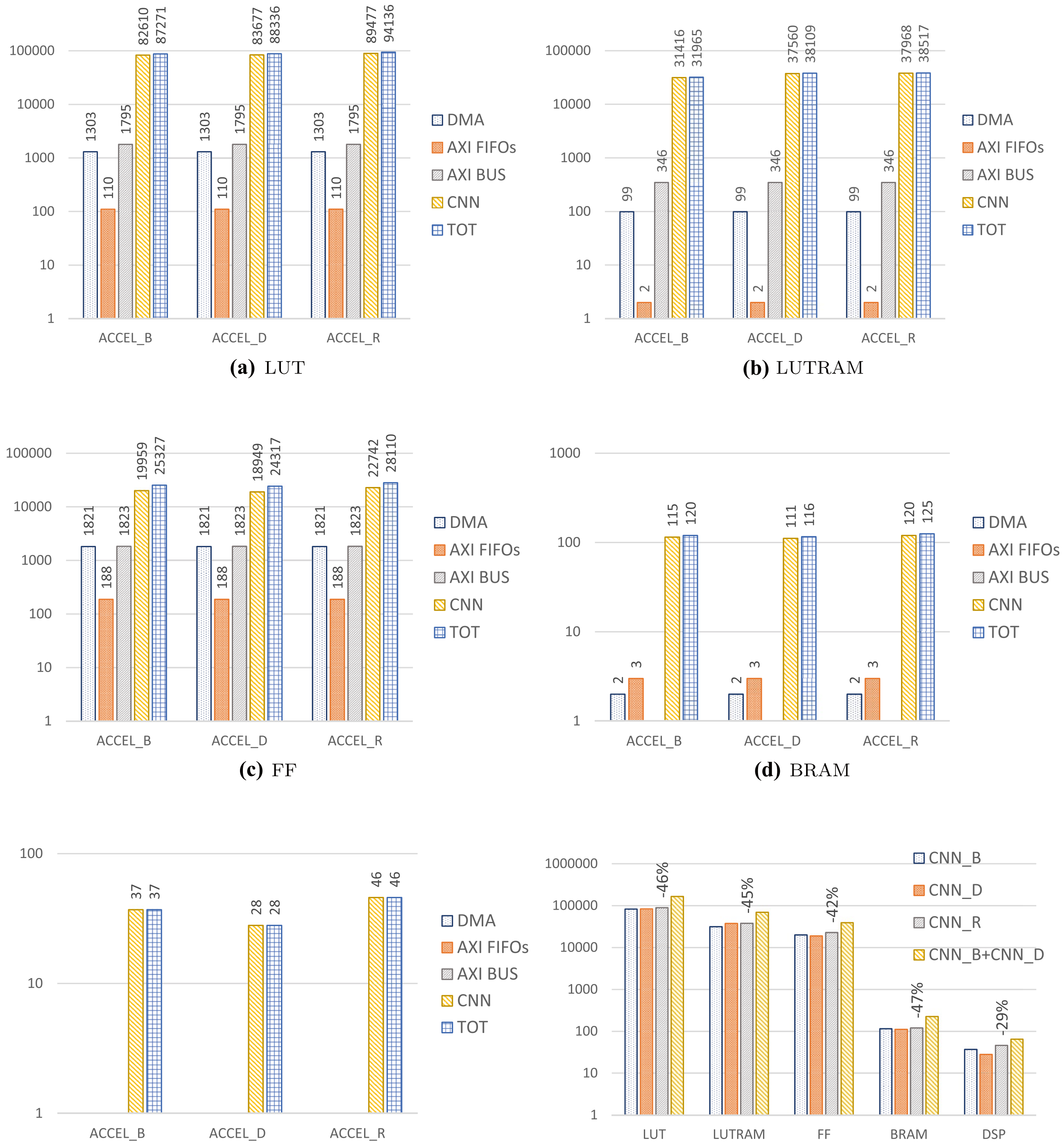


Figure 8 Overview of the overall accelerator architecture.



(a) LUT

(b) LUTRAM

(c) FF

(d) BRAM

(e) DSP

(f) Overall

Figure 9 Resource occupancy of the systems implementing the CNN accelerator. In Fig. 9f there are the overall CNN accelerators, where labels indicate percentage variation of CNN_R with respect to the sum of CNN_B and CNN_D.

Execution Time Dealing with the execution time related to a single image classification on the three considered designs, results are reported on Table 4. The resulting overhead of convolutional actors implemented by *CNN_D* with respect to the baseline *CNN_B* is about 8%. A trade-off is then present as execution time is lower in *CNN_B*, the most resource-consuming accelerator. Indeed, the resource-saving introduced in *CNN_D* by mapping two layers in a single actor (see Section 3.3) is paid with a degradation of the execution time.

Table 4 Execution time for image classification. % indicate percentage variation with respect to *ACCEL_B*.

Accelerator	Execution time [ms]	%
ACCEL_B	538	-
ACCEL_D	583	8
ACCEL_R	538 - 583*	0 - 8*

*Execution time depends on the configuration

4.2 Focus on Single Actors and Layers

In this section, a detail on single actors and layers results is proposed, in order to better understand what is happening within the implemented CNN, how adaptivity is reached in the practice, and what it implies on the system performance. In this case, data have been collected with Xilinx Vivado tools: synthesis, simulation, and power estimation (considering switching activity gathered during post-synthesis simulations of the designs). The operating frequency is 100 MHz in all cases.

During the analysis of single convolutional actors and layers, three different dimensions are considered to highlight the relationship of the analyzed metrics with the size of the specific actors and layers. In particular, the designs under test will be:

- *small*, with 8x3x1x1 convolutional actors and layers;
- *medium*, with 16x64x1x1 convolutional actors and layers;
- *large*, with 64x32x3x3 convolutional actors and layers.

These three actors and layers have been developed considering both baseline (*CONV_B*, *LAYER_B*) and double (*CONV_D*, *LAYER_D*) versions of the convolutional actors described in Section 3.3. Here *CONV_B* and *CONV_D* refer to the only red and blue circles respectively in Figs. 5 and 6, while *LAYER_B* and *LAYER_D* to all the resources involved in the same Figures (so that including FIFOs, weight, bias, and line buffer actors, besides the convolutional actors). Additionally, *2xCONV_B* and *2xLAYER_B* designs, which are simply a sum of two baseline actors and layers, will be considered to fairly compare *CONV_D* and *LAYER_D*, which actually implement two actors and layers.

Resource utilization Resource occupancy data for single actors and layers are shown in Table 5. Double versions (*CONV_D* and *LAYER_D*), as expected, present an increase of resources with respect to the corresponding baseline ones (*CONV_B* and *LAYER_B*), as depicted in column $\%_1$. Such increase is always below 35% in the case of single actors (*CONV_B* versus *CONV_D*), while it is larger, even more than 100%, in case of single layers (*LAYER_B* versus *LAYER_D*). This difference comes from the fact that single layers, besides the convolutional actor, also involve line buffers, weight and bias actors, and FIFOs. Thus, since *LAYER_D* implements two convolutional layers, it includes line buffers, weights, bias, and FIFOs of corresponding to two different layers. For this reason, a more fair comparison is given in column $\%_2$, where double versions are compared with two parallel baseline ones (the previously mentioned *2xCONV_B* and *2xLAYER_B*). Compared to these functionally equivalent designs, *CONV_D* and *LAYER_D* are almost always employing less resources, reaching peaks of 50% of saving in some cases. LUTRAMs in *LAYER_D* are the only resources that are not saved with respect to *2xLAYER_B*. The reason is that this type of resource is used by the synthesizer only to realize the FIFOs, and the number of FIFOs required by *CONV_D* is exactly double the one in *CONV_B*, and they have all the same number of slots.

Dynamic Power Consumption Dynamic power consumption data for single actors and layers are shown in Table 6. Here, resource occupancy evidence is reflected since power consumption is directly proportional to the number of employed logic. In particular, the performance of baseline and double designs is the same in terms of single actors (*CONV_B* versus *CONV_D*), meaning that double actors consume about the same power as baseline ones. This absence of diversity is mainly due to the fact that the designs are too small to appreciate a variation in the power estimation. Please note that, if two parallel baseline actors are considered as a fairer term of comparison (column $\%_2$) than one unique baseline actor (column $\%_1$), double actors anyway behave better than baseline ones. The difference is, instead, evident when entire layers are considered (*LAYER_B* versus *LAYER_D*). Going from baseline to double versions, an increase of consumed power is still present, as in terms of resources, and the increase is in some cases very significant, as for clk contribution. However, as occurred for resources, when the comparison with *2xLAYER_B* is considered (column $\%_2$), the saving is quite clear, reaching 50% and more in different terms. Overall, looking at the dyn term, which is the sum of all the others, the *LAYER_D* saves always more than 30% of power with respect to *2xLAYER_B*.

Execution Time While in terms of resources and power consumption the benefits of double versions have emerged, the drawbacks of computing two convolutional layers with the same logic, as seen in the previous section, lie in execution latency. In Table 7 execution latency of the different considered layers is reported. Such data refer to a test condition where each layer is fed with an entire tensor necessary for a single image classification of the CNN, and input/output are always readable/writable, that is empty/full conditions in incoming/outgoing FIFOs never occur. Please note that, in such test condition, *CONV_B* and *LAYER_B* execution latency is almost the same, so that only *LAYER_B* data

Table 5 Resource occupancy of convolutional actors/layers. # indicates the number of used slices. $\%_1$ and $\%_2$ indicate percentage variation with respect to *CONV_B*/*LAYER_B* and *2xCONV_B*/*2xLAYER_B* respectively, these latter corresponding to two times the power of the former.

design	**resource**	*small*			*medium*			*large*		
		#	$\%_1$	$\%_2$	#	$\%_1$	$\%_2$	#	$\%_1$	$\%_2$
CONV_B	LUT	73	-	-	80	-	-	101	-	-
	LUTRAM	0	-	-	0	-	-	0	-	-
	FF	88	-	-	98	-	-	110	-	-
	BRAM	0.5	-	-	0.5	-	-	0.5	-	-
	DSP	1	-	-	1	-	-	1	-	-
CONV_D	LUT	97	+33	-34	101	+26	-37	120	+19	-41
	LUTRAM	0	+0	+0	0	+0	+0	0	+0	+0
	FF	107	+22	-39	116	+18	-41	126	+15	-43
	BRAM	0.5	+0	-50	0.5	+0	-50	0.5	+0	-50
	DSP	1	+0	-50	1	+0	-50	1	+0	-50
LAYER_B	LUT	500	-	-	740	-	-	3218	-	-
	LUTRAM	80	-	-	80	-	-	80	-	-
	FF	569	-	-	639	-	-	717	-	-
	BRAM	1	-	-	2.5	-	-	2	-	-
	DSP	1	-	-	1	-	-	2	-	-
LAYER_D	LUT	982	+96	-2	1454	+96	-2	6357	+98	-1
	LUTRAM	160	+100	+0	160	+100	+0	160	+100	+0
	FF	1083	+90	-5	1214	+90	-5	1340	+87	-7
	BRAM	1.5	+50	-25	4.5	+80	-10	3.5	+75	-13
	DSP	1	+0	-50	1	+0	-50	3	+50	-25

are depicted in the Table 7. Double versions of the layers (*LAYER_D*) take exactly double the time than baseline ones (*LAYER_B*) to process the input tensor, for all the three considered dimensions of the kernel. This is true either for the single baseline layer case (column $\%_1$) and for the parallel baseline layer one (column $\%_2$) since, being parallel, the two layers are executed simultaneously in these latter designs. So, resources and power savings of the double designs seen in Tables 5 and 6 are paid by doubling the execution time of the layers.

In the end, the proposed solutions can be further improved by exploiting the presented toolchain: more than two working points can be derived to have a wider adaptivity response (that is modeling more dataflows in step *N1* of Fig. 3), performance can be further enhanced by using other pragmas, e.g. total or partial loop unrolling, on the convolutional actors during the HLS step (*A2* of Fig. 3) as well as on the other actors, other metrics (e.g. accuracy of the CNN classification) can be involved in the trade-offs achieved among working points, thus acting on both steps *N1* and *A2*, the ones enabling adaptivity shaping.

4.3 Evaluation Summary

At the single actor and layer level, the trade-off between power consumption and execution latency due to the adoption of the same resources for performing two convolutional layers in the double version of the CNN is clearly visible, since overall dynamic power is reduced by 33%, 31%, 44% respectively in small, medium, and large layer design (Table 6 cols. $\%_2$) at the price of doubling the execution time in every layer design (Table 7 cols. $\%_2$). However, energy is commonly considered more important than power in the addressed context. We can calculate the energy consumption of an image classification by multiplying the power consumption by the execution time. To see an energy/execution-time trade-off between the two proposed configurations, the overall-accelerator data must be considered (Section 4.1). In this case, the difference in execution time for using the *LAYER_D* layers is only 8% more than the single *LAYER_B* layers (Table 4), leading to an expected saving in energy when this configuration is running. At this point, power measurements on the accelerators running on board have

Table 6 Dynamic power consuption in mW of convolutional actors/layers (columns mW). $\%_1$ and $\%_2$ indicate percentage variation with respect to *CONV_B*/*LAYER_B* and *2xCONV_B*/*2xLAYER_B* respectively, these latter corresponding to two times the resources of the former. dyn refers to the total dynamic power, the sum of the other terms: clock (clk), signal (sig), logic (logic), BRAM (bram), and DSP (dsp).

design	**power**	*small*			*medium*			*large*		
		mW	$\%_1$	$\%_2$	mW	$\%_1$	$\%_2$	mW	$\%_1$	$\%_2$
CONV_B	dyn	2	-	-	3	-	-	4	-	-
	clk	0	-	-	0	-	-	0	-	-
	sig	0	-	-	1	-	-	1	-	-
	log	0	-	-	1	-	-	1	-	-
	bram	1	-	-	1	-	-	1	-	-
	dsp	0	-	-	0	-	-	0	-	-
CONV_D	dyn	2	+0	-50	3	+0	-50	4	+0	-50
	clk	0	+0	+0	0	+0	+0	0	+0	+0
	sig	0	+0	+0	1	+0	-50	1	+0	-50
	log	0	+0	+0	1	+0	-50	1	+0	-50
	bram	1	+0	-50	1	+0	-50	2	+0	-50
	dsp	0	+0	+0	0	+0	+0	0	+0	+0
LAYER_B	dyn	6	-	-	8	-	-	26	-	-
	clk	2	-	-	2	-	-	3	-	-
	sig	1	-	-	2	-	-	9	-	-
	log	1	-	-	2	-	-	13	-	-
	bram	1	-	-	2	-	-	2	-	-
	dsp	0	-	-	1	-	-	1	-	-
LAYER_D	dyn	8	+33	-33	11	+38	-31	29	+12	-44
	clk	4	+100	+0	4	+100	+0	5	+67	-17
	sig	1	+0	-50	2	+0	-50	8	-11	-56
	log	1	+0	-50	2	+0	-50	13	+0	-50
	bram	1	+0	-50	2	+0	-50	2	+50	-25
	dsp	0	+0	+0	0	+0	+0	0	+0	+0

Table 7 Execution latency in ms of convolutional layers (columns ms). $\%_1$ and $\%_2$ indicate percentage variation with respect to *LAYER_B* and *2xLAYER_B* respectively, these latter corresponding to the same latency of the former (it is two *LAYER_B* in parallel).

design	*small*			*medium*			*large*		
	ms	$\%_1$	$\%_2$	ms	$\%_1$	$\%_2$	ms	$\%_1$	$\%_2$
LAYER_B	9.8	-	-	51.9	-	-	85.9	-	-
LAYER_D	19.6	+100	+100	103.8	+100	+100	171.8	+100	+100

not been carried out yet. Having proved with this preliminary paper the feasibility and potential benefits of supporting reconfiguration, we will explore in depth the trade-off execution, as discussed in Section 5.2.

5 Conclusions

Efficiency and flexible behaviors have nowadays turned out to be a must in many application domains when cyber-physical entanglement is there. Designers cope with it by leveraging on heterogeneous systems, whose design still requires a lot of expertise, especially when custom computation units have to be derived. This for sure does not help in terms of design time and development costs so investments and studies related to design automation and frameworks are still on the hype.

All these considerations are certainly true in the case of image and video processing at the edge, especially considering classification tasks using neural networks. This is a key aspect in cyber-physical systems, as context awareness is crucial to take correct decisions and performing actions. The scientific community is quite active in providing new, more efficient implementations and tools for neural networks targeting different devices and features. However, dealing with HW accelerators it is not yet capable of delivering full support for neural networks delivering high execution efficiency, but also flexibility through adaptive behaviors. But, still in the context of cyber-physical systems, adapting execution metrics to the ongoing situation is necessary to meet conflicting performance constraints.

5.1 Summary

In this work, we have presented a novel toolchain for aiding developers in modeling neural networks, deriving almost automatically the corresponding HW accelerator, and giving to the user the possibility of molding on top of that a certain degree of adaptivity. The proposed toolchain requires an ONNX neural network model as input, it adopts two open-source tools, the ONNXparser, and the Multi-Dataflow Composer, to derive a lower-level specification of the same network. Adaptivity can be shaped by the user on the same input ONNX model(s) or on the lower specification of the network, according to the specific needs, knowledge, and skills. In the test case proposed in this work, Vivado HLS is then applied to derive final HW specifications for the accelerator, but in theory, other HLS engines can be adopted, open to the possibility of addressing target devices different from FPGAs, targeted by Vivado HLS.

A CNN for humans/animals classification, used within a FitOptiVis project Use Case regarding a critical infrastructure surveillance scenario, is adopted as a proof of concept. Such CNN is made adaptable by deriving, at the Vivado HLS input stage, logic employing different amounts of resources for elaborating the convolutional layers. As a result, the CNN accelerator can adapt its behavior according to the context needs: if the battery is running out, it can change profile and consume about 30% less power in each layer at the price of an extra 8% time to classify an image; on the contrary, if response time is crucial, e.g. due to preliminary alarms already raised, the network could be executed at the maximum speed by pulling more energy from the battery. Note that the obtained solution is purely demonstrative, to show the potentials of the proposed flow, so the final implementation is not optimized and is not competitive with respect to the state-of-the-art.

5.2 Future work

Being this a demonstrative work to show the effectiveness of exploiting runtime reconfigurability to design adaptive accelerators for CNNs, several further improvements are planned and ongoing.

Design Flow While maintaining the same structure, the toolchain will be adapted to be compatible with alternative HLS engines, that in turn can enlarge the set of target devices and available features. Moreover, the user intervention will be reduced in manual steps where adaptivity is described (steps *N1* and *A2*). The mapping from the ONNX model to the dataflow model (step *N1*) will be automated, but the user will necessarily still be in charge of designing the ONNX model and shaping the adaptivity. The code refactoring and pragma insertion (step can be *A2*) will be partially automated through the extension of the ONNXparser. Most effective pragmas can be automatically utilized, e.g. loop pipelining. While the insertion/tuning of other ones, e.g. loop unrolling, can be facilitated, leaving to the user the duty of selecting the desired trade-off, e.g. the unrolling factor.

Adaptivity Experiments Adaptivity support has been demonstrated on a power-vs-latency tradeoff and considering a limited part of the system (two convolutional layers). A more complete study with accurate measurements, taking into consideration also the switching among contexts/scenarios, is planned. Such measurements will be taken directly on the system running on the physical target board and will reveal the possibility of achieving an energy-vs-latency tradeoff, not known at this development point. Moreover, adaptivity support will be investigated by targeting bigger/deeper CNN models as well as different kinds of tradeoffs, e.g. trading off data precision with energy consumption. The integration of the adaptive accelerators with an embedded OS is ongoing and will serve as a first step to developing a more accurate and easy-to-use testing environment for the proposed design flow.

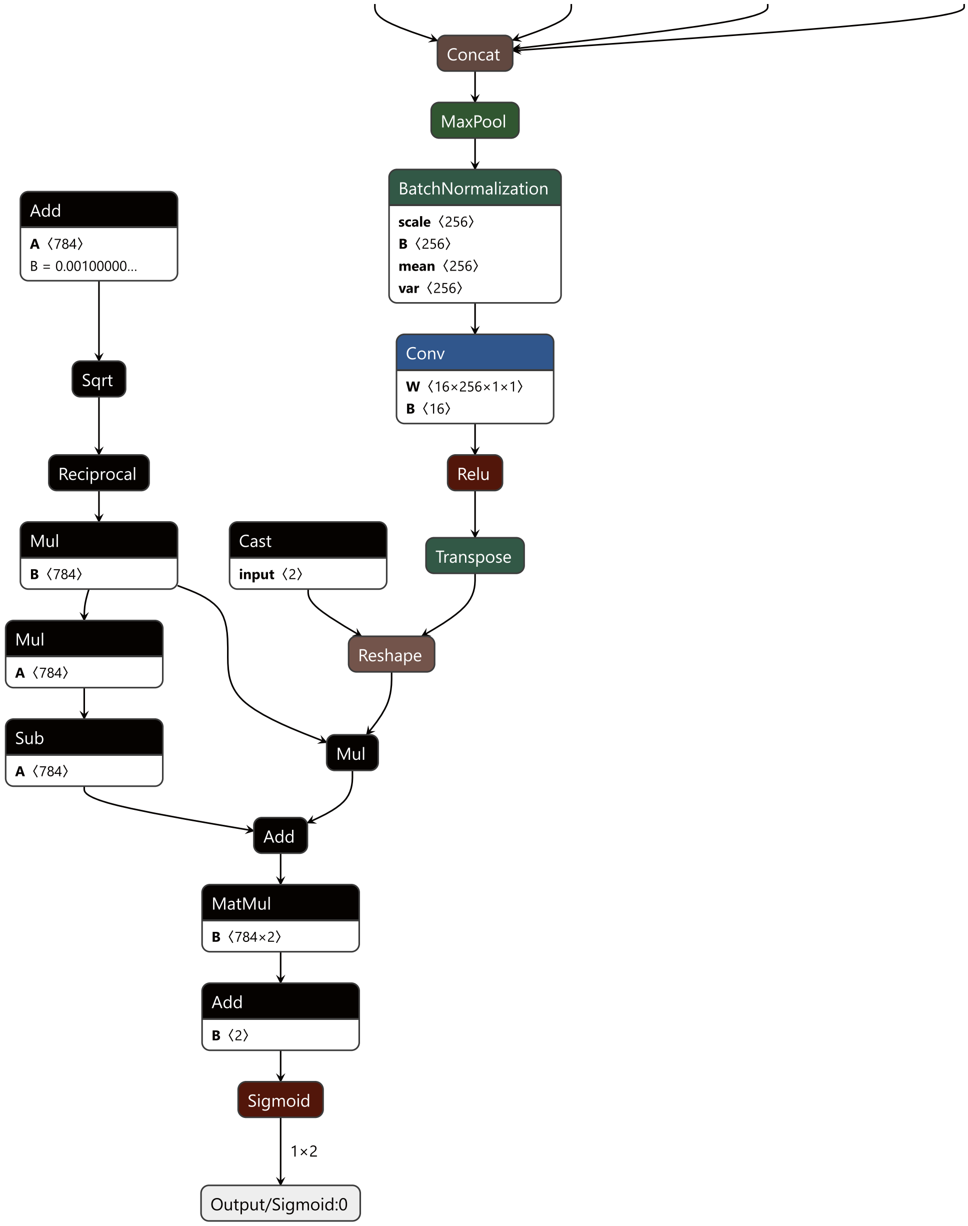
Appendix. Modelling the Networks with Keras

The starting network given by the use case provider of the FitOptiVis project was already in ONNX format, and it was derived from an original Keras model. However, several incompatibilities with respect to the adopted ONNX-parser have emerged, including some layers that are not supported by ONNXparser, such as *Cast*, *Sqrt* and *Transpose* (all indicated as black modules in Fig. 10). Moreover, by the initial analysis, it turned out that other optimizations were needed, like the substitution of the cascade of *Cast+Reshape* into a *Flatten* and of the *Matmul+Add* into a *Gemm* (again, all indicated as black modules in Fig. 4).

The tf2onnx [47] library has been originally used to export the networks. Taking into account the characteristics of the used converter, here follows the list of detected issues that led to the incompatibilities that should be avoided/overcome:

- Transpose: TensorFlow uses NHWC data format by default, while ONNX uses NCWH. This layer is needed to overcome the data format incompatibility when trans-

Figure 10 InceptionNet structure generated from Keras: black modules highlight the main incompatibilities with the adopted ONNXparser.



lating the network from one framework to another, but its implementation is not supported by the converter [48].

- Exploded Dense Layer (*gemm* = *matmul*+*add*): the optimized representation of the Dense layer in the ONNX model is through Gemm operator[5]. However, it was not supported by tf2onnx until May 14th 2019, but also the latest version of the converter is still reporting problems on this topic[6].
- Flatten: the correct conversion should be *constant* + *reshape*, since Cast operation is not accepted by the ONNXparser. Until now, all Pytorch to ONNX converters are able to optimize it directly into a *flatten* operator, except for the TensorFlow to ONNX ones[7].
- Exploded Batch Normalization (*Sqrt*, *Reciprocal*, *Sub*, *Mul*, *Add*): from the carried out analysis, this is caused by the *Batch Normalization* operator inserted right after

[5] https://github.com/onnx/onnx/issues/1682

[6] https://github.com/onnx/tensorflow-onnx/issues/516#issuecomment-509953151

[7] https://github.com/onnx/tensorflow-onnx/issues/490#issuecomment-497270831

Flatten in the provided net. his causes the division of Batch Normalization in multiple sub-layers, some of them, such as *Sqrt* and *Reciprocal*, are not supported by the converter.

Therefore, a more suitable converter for the input models had to be searched for. As a first candidate, the keras2onnx [49] converter has been considered. However, although the conversion resulted improved, there were some remaining issues: non supported *identity* and *traspose* functions were there, as well as the already reported issue regarding the Exploded Dense Layer. Then, the new ONNXMLTools library [50] has been tested. This library includes a wrapper of keras2onnx, and it allows to take the non-compatible conversion from Keras to ONNX as input, making it possible to apply an optimizer to eliminate unnecessary layers or to merge two or more layers into one. Using the result obtained with keras2onnx, the following optimizations have been applied: `eliminate_identity` and `fuse_matmul_add_bias_into_gemm`. Nevertheless, an unsolved bug[8] in the ONNX library code used by ONNXMLTools prevented the network from being optimized. It seems[9] that, at the moment, the only way to use this type of optimizer without problems is starting from a network described with Pytorch framework instead of Keras.

In the end, this is what has been done in this work, where InceptionNet has been re-trained with Pytorch. Nevertheless, we do not exclude that the proposed flow with different starting models could be compatible with Keras too, provided that there is also a constant effort in optimizing the aforementioned libraries and converters.

Funding Open access funding provided by Università degli Studi di Sassari within the CRUI-CARE Agreement. This work is part of the FitOptiVis project [2], funded by the ECSEL Joint Undertaking under grant number H2020-ECSEL-2017-2-783162, and of the Comp4Drones project No. 826610, ECSEL-JU 2018. This work was also partly supported by the EU H2020 project ALOHA, under the European Union's Horizon 2020 research and innovation programme (grant no. 780788).

[8] https://github.com/onnx/onnx/issues/2417#issue-512130707

[9] https://github.com/onnx/onnx/issues/1385#issuecomment-545773838

References

1. An FPGA "Companion" in Smartphone Design - A Lattice Semiconductor White Paper. Document ID 47335 (2012-05).
2. Al-Ars, Z., et al. (2019). The fitoptivis ECSEL project: highly efficient distributed embedded image/video processing in cyber-physical systems. In *Conference on Computing Frontiers* (pp. 333–338).
3. Pomante, L., Palumbo, F., Rinaldi, C., Valente, G., Sau, C., Fanni, T., van der Linden, F., Basten, T., Geilen, M., Peeren, G., Kadlec, J., Jääskeläinen, P., de Alejandro, M. M., Saarinen, J., Säntti, T., Zedda, M. K., Sanchez, V., Goswami, D., Al-Ars, Z., & de Beer, A. (2020). Design and management of image processing pipelines within CPS: 2 years of experience from the fitoptivis ECSEL project. In *23rd Euromicro Conference on Digital System Design, DSD 2020, Kranj, Slovenia, August 26-28, 2020* (pp. 378–385). IEEE. Retrieved April 2023, from https://www.latticesemi.com/-/media/LatticeSemi/Documents/WhitePapers/AG/AnFPGACompanioninSmartphoneDesign.ashx?document_id=47335
4. Beaumin, C., Sentieys, O., Casseau, E., & Carer, A. (2010). A coarse-grain reconfigurable hardware architecture for RVC-CAL-based design. In *2010 Conference on Design and Architectures for Signal and Image Processing (DASIP)* (pp. 152–159). https://doi.org/10.1109/DASIP.2010.5706259
5. Fanni, T., Rodríguez, A., Sau, C., Suriano, L., Palumbo, F., Raffo, L., & de la Torre, E. (2018). Multi-grain reconfiguration for advanced adaptivity in cyber-physical systems. In *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)* (pp. 1–8). https://doi.org/10.1109/RECONFIG.2018.8641705
6. Wijtvliet, M., Waeijen, L., & Corporaal, H. (2016). Coarse grained reconfigurable architectures in the past 25 years: Overview and classification. In W. A. Najjar, & A. Gerstlauer (Eds.), *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, SAMOS 2016, Agios Konstantinos, Samos Island, Greece, July 17-21, 2016* (pp. 235–244). IEEE.
7. Sau, C., Palumbo, F., Pelcat, M., Heulot, J., Nogues, E., Ménard, D., Meloni, P., & Raffo, L. (2017). Challenging the best HEVC fractional pixel FPGA interpolators with reconfigurable and multifrequency approximate computing. *IEEE Embed. Syst. Lett.*, *9*(3), 65–68.
8. Palumbo, F., Sau, C., Fanni, T., & Raffo, L. (2017). Challenging CPS trade-off adaptivity with coarse-grained reconfiguration. In A. D. Gloria (ed.) *Applications in Electronics Pervading Industry, Environment and Society - APPLEPIES 2017, Rome, Italy, 21-22 September 2017, extitLecture Notes in Electrical Engineering* (vol. 512, pp. 57–63). Springer.
9. Diniz, C. M., Shafique, M., Bampi, S., & Henkel, J. (2015). A reconfigurable hardware architecture for fractional pixel interpolation in high efficiency video coding. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 34*(2), 238–251. https://doi.org/10.1109/TCAD.2014.2384517
10. Coutinho Demetrios, A. M., De Sensi, D., Lorenzon, A. F., Georgiou, K., Nunez-Yanez, J., Eder, K., Xavier-de Souza, S. (2020). Performance and energy trade-offs for parallel applications on heterogeneous multi-processing systems. *Energies*, *13*(9).
11. Suriano, L., Otero, A., Rodríguez, A., Sánchez-Renedo, M., & La Torre, E. D. (2020). Exploiting multi-level parallelism for run-time adaptive inverse kinematics on heterogeneous mpsocs. *IEEE Access, 8*, 118707–118724. https://doi.org/10.1109/ACCESS.2020.3005202
12. Mittal, S. (2014). A survey of techniques for improving energy efficiency in embedded computing systems. *Int. J. Comput. Aided Eng. Technol.*, *6*(4), 440–459.
13. Guo, K., Zeng, S., Yu, J., Wang, Y., & Yang, H. (2019). [dl] a survey of FPGA-based neural network inference accelerators.

ACM Transactions on Reconfigurable Technology and Systems (TRETS), 12(1), 1–26.

14. Xilinx Vitis AI development environment. https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html
15. Xilinx Deep Learning Processing Unit. https://www.xilinx.com/products/intellectual-property/dpu.html
16. Yu, X., Wang, Y., Miao, J., Wu, E., Zhang, H., Meng, Y., Zhang, B., Min, B., Chen, D., & Gao, J. (2019). A data-center FPGA acceleration platform for convolutional neural networks. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)* (pp. 151–158). IEEE.
17. Zhang, C., Fang, Z., Zhou, P., Pan, P., & Cong, J. (2016). Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (pp. 1–8). https://doi.org/10.1145/2966986.2967011
18. Ma, Y., Cao, Y., Vrudhula, S., & S. Seo, J. (2017). An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)* (pp 1–8). https://doi.org/10.23919/FPL.2017.8056824
19. Venieris, S. I., & Bouganis, C. S. (2017). Latency-driven design for FPGA-based convolutional neural networks. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)* (pp. 1–8). https://doi.org/10.23919/FPL.2017.8056828
20. Gokhale, V., Zaidy, A., Chang, A. X. M., Culurciello, E. (2017). Snowflake: An efficient hardware accelerator for convolutional neural networks. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)* (pp. 1–4). https://doi.org/10.1109/ISCAS.2017.8050809
21. Qiu, J., Wang, J., Yao, S., Guo, K., Li, B., Zhou, E., Yu, J., Tang, T., Xu, N., Song, S., Wang, Y., & Yang, H. (2016). Going deeper with embedded FPGA platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '16* (pp. 26–35). ACM, New York, NY, USA. https://doi.org/10.1145/2847263.2847265. http://doi.acm.org/10.1145/2847263.2847265
22. Meloni, P., Capotondi, A., Deriu, G., Brian, M., Conti, F., Rossi, D., Raffo, L., & Benini, L. (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). https://doi.org/10.1145/3284357
23. Carreras, M., Deriu, G., Raffo, L., Benini, L., & Meloni, P. (2020). Optimizing temporal convolutional network inference on FPGA-based accelerators. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems, 10*(3), 348–361. https://doi.org/10.1109/JETCAS.2020.3014503
24. Prost-Boucle, A., Bourge, A., Petrot, F., Alemdar, H., Caldwell, N., & Leroy, V. (2017). Scalable high-performance architecture for convolutional ternary neural networks on FPGA. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)* (pp. 1–7). https://doi.org/10.23919/FPL.2017.8056850
25. Umuroglu, Y., Fraser, N., Gambardella, G., Blott, M., Leong, P., Jahre, M., & Vissers, K. (2017). Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '17* (pp. 65–74). ACM, New York, NY, USA. https://doi.org/10.1145/3020078.3021744. http://doi.acm.org/10.1145/3020078.3021744
26. Rasoulinezhad, S., Zhou, H., Wang, L., & Leong, P. H. W. (2019). PIR-DSP: An FPGA DSP block architecture for multi-precision deep neural networks. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (pp. 35–44).
27. Wang, E., Davis, J. J., Cheung, P. Y., & Constantinides, G. (2020). Lutnet: Learning FPGA configurations for highly efficient neural network inference. *IEEE Transactions on Computers*.
28. Meloni, P., Loi, D., Deriu, G., Pimentel, A. D., Sapra, D., Moser, B., Shepeleva, N., Conti, F., Benini, L., Ripolles, O., Solans, D., Pintor, M., Biggio, B., Stefanov, T. P., Minakova, S., Fragoulis, N., Theodorakopoulos, I., Masin, M., & Palumbo, F. (2018). ALOHA: an architectural-aware framework for deep learning at the edge. In M. Martina, & W. Fornaciari (Eds.), *Proceedings of the Workshop on INTelligent Embedded Systems Architectures and Applications, INTESA@ESWEEK 2018, Turin, Italy, October 04-04, 2018* (pp. 19–26). ACM.
29. Meloni, P., Loi, D., Deriu, G., Carreras, M., Conti, F., Capotondi, A., & Rossi, D. (2019). Exploring Neuraghe: A customizable template for APSOC-based CNN inference at the edge. *IEEE Embedded Systems Letters, PP*, 1–1. https://doi.org/10.1109/LES.2019.2947312
30. Nane, R., Sima, V., Pilato, C., Choi, J., Fort, B., Canis, A., Chen, Y. T., Hsiao, H., Brown, S., Ferrandi, F., Anderson, J., & Bertels, K. (2016). A survey and evaluation of FPGA high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 35*(10), 1591–1604. https://doi.org/10.1109/TCAD.2015.2513673
31. Pursley, D., & Yeh, T. (2017). High-level low-power system design optimization. In *Symposium on VLSI Design, Automation and Test* (pp. 1–4).
32. Xilinx: Vivado Design Suite User Guide - High-Level Synthesis, UG902
33. Rubattu, C., Palumbo, F., Sau, C., Salvador, R., Sérot, J., Desnos, K., Raffo, L., & Pelcat, M. (2019). Dataflow-functional high-level synthesis for coarse-grained reconfigurable accelerators. *IEEE Embedded Systems Letters, 11*(3), 69–72. https://doi.org/10.1109/LES.2018.2882989
34. Compton, K., & Hauck, S. (2002). Reconfigurable computing: A survey of systems and software. *34*(2), 171–210. https://doi.org/10.1145/508352.508353
35. Fanni, T., Sau, C., Raffo, L., & Palumbo, F. (2015). *Automated power gating methodology for dataflow-based reconfigurable systems*. https://doi.org/10.1145/2742854.2747285. Cited By 9
36. Li, L., Sau, C., Fanni, T., Li, J., Viitanen, T., Christophe, F., Palumbo, F., Raffo, L., Huttunen, H., Takala, J., Bhattacharyya, S. S. (2019). An integrated hardware/software design methodology for signal processing systems. *Journal of Systems Architecture, 93*, 1–19. https://doi.org/10.1016/j.sysarc.2018.12.010, https://www.sciencedirect.com/science/article/pii/S1383762118301735
37. Bezati, E., Casale-Brunet, S., Mosqueron, R., & Mattavelli, M. (2019). An heterogeneous compiler of dataflow programs for ZYNQ platforms. In *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (pp. 1537–1541). https://doi.org/10.1109/ICASSP.2019.8682525
38. Sérot, J., & Berry, F. (2014). High-level dataflow programming for reconfigurable computing. In *Symposium on Computer Architecture and High Performance Computing Work* (pp. 72–77).
39. Sau, C., Fanni, T., Rubattu, C., Raffo, L., & Palumbo, F. (2021). The multi-dataflow composer tool: An open-source tool suite for optimized coarse-grain reconfigurable hardware accelerators and platform design. *Microprocessors and Microsystems*, *80*, 103326. https://doi.org/10.1016/j.micpro.2020.103326, https://www.sciencedirect.com/science/article/pii/S0141933120304853
40. Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2015). Rethinking the inception architecture for computer vision. CoRR abs/1512.00567.
41. Sau, C., Rinaldi, C., Pomante, L., Palumbo, F., Valente, G., Fanni, T., Martinez, M., van der Linden, F., Basten, T., Geilen, M., Peeren, G., Kadlec, J., Jääskeläinen, P., Bulej, L., Barranco, F., Saarinen, J., Säntti, T., Zedda, M.K., Sanchez, V., Nikkhah, S.T.,

Goswami, D., Amat, G., Maršík, L., van Helvoort, M., Medina, L., Al-Ars, Z., & de Beer, A. (2021). Design and management of image processing pipelines within CPS: Acquired experience towards the end of the Fitoptivis Ecsel project. *Microprocessors and Microsystems*, *87*.

42. Agarap, A. F. (2018). Deep learning using rectified linear units (RELU). CoRR abs/1803.08375. http://arxiv.org/abs/1803.08375
43. PyTorch: Batchnorm2d (2019). https://pytorch.org/docs/stable/generated/torch.nn.BatchNorm2d.html
44. Ding, B., Qian, H., & Zhou, J. (2018). Activation functions and their characteristics in deep neural networks.
45. Zhang Zhifei, S. Y., & Qi, H. (2017). Age progression/regression by conditional adversarial autoencoder. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE.
46. Corrado, A. (2019). *Animal-10 dataset*. https://www.kaggle.com/alessiocorrado99/animals10
47. Bai, J., Lu, F., Zhang, K., et al. (2019). Onnx: Open neural network exchange. Retrieved April 2023, from https://docs.xilinx.com/v/u/en-US/ug902-vivado-high-level-synthesis, https://github.com/onnx/onnx
48. TensorFlow: Tensorflow core v2.4.1 (2020). https://www.tensorflow.org/api_docs/python/tf/nn/conv2d
49. Keras ONNX: Keras to ONNX converter (2021). https://github.com/onnx/keras-onnx
50. ONNxmltools: ONNxmltools library (2021). https://github.com/onnx/onnxmltools