This is the peer reviewd version of the followng article:

Neuraghe: Exploiting CPU-FPGA synergies for efficient and flexible CNN inference acceleration on zynQ SoCs / Meloni, P., Capotondi, A., Deriu, G., Brian, M., Conti, F., Rossi, D., Raffo, L., Benini, L.. - In: ACM TRANSACTIONS ON RECONFIGURABLE TECHNOLOGY AND SYSTEMS. - ISSN 1936-7406. - 11:3(2018), pp. 1-24. [10.1145/3284357]

Terms of use:

The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

11/10/2026 13:31

(Article begins on next page)

This is the accepted manuscript of:

P. Meloni, A. Capotondi, G. Deriu, M. Brian, F. Conti, D. Rossi, L. Raffo and L. Benini (2018) NEURAghe: Exploiting CPU-FPGA Synergies for Efficient and Flexible CNN Inference Acceleration on Zynq SoCs. ACM Trans Reconfigurable Technol Syst. 11(3):18:1–18:24. doi: 10.1145/3284357

© ACM 2018. This version of the work is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in https://doi.org/10.1145/3284357

NEURAghe: Exploiting CPU-FPGA Synergies for Efficient and Flexible CNN Inference Acceleration on Zynq SoCs

PAOLO MELONI, Università di Cagliari, Italy
ALESSANDRO CAPOTONDI, Università di Bologna, Italy
GIANFRANCO DERIU, Università di Cagliari, Italy and T3LAB, Italy
MICHELE BRIAN, T3LAB, Italy
FRANCESCO CONTI, Università di Bologna, Italy and ETH Zurich, Switzerland
DAVIDE ROSSI, Università di Bologna, Italy
LUIGI RAFFO, Università di Cagliari, Italy
LUCA BENINI, Università di Bologna, Italy and ETH Zurich, Switzerland

Deep convolutional neural networks (CNNs) obtain outstanding results in tasks that require human-level understanding of data, like image or speech recognition. However, their computational load is significant, motivating the development of CNN-specialized accelerators. This work presents NEURAghe, a flexible and efficient hardware/software solution for the acceleration of CNNs on Zynq SoCs. NEURAghe leverages the synergistic usage of Zynq ARM cores and of a powerful and flexible Convolution-Specific Processor deployed on the reconfigurable logic. The Convolution-Specific Processor embeds both a convolution engine and a programmable soft core, releasing the ARM processors from most of the supervision duties and allowing the accelerator to be controlled by software at an ultra-fine granularity. This methodology opens the way for cooperative heterogeneous computing: while the accelerator takes care of the bulk of the CNN workload, the ARM cores can seamlessly execute hard-to-accelerate parts of the computational graph, taking advantage of the NEON vector engines to further speed up computation. Through the companion NeuDNN SW stack, NEURAghe supports end-to-end CNN-based classification with a peak performance of 169 GOps/s, and an energy efficiency of 17 GOps/W. Thanks to our heterogeneous computing model, our platform improves upon the state-of-the-art, achieving a frame rate of 5.5 fps on the end-to-end execution of VGG-16, and 6.6 fps on ResNet-18.

CCS Concepts: • **Computer systems organization → Embedded systems**; • **Hardware → Reconfigurable logic and FPGAs**; **Hardware accelerators**; • **Computing methodologies** → *Computer vision problems*;

Additional Key Words and Phrases: FPGAs, Convolutional Neural Networks, HW accelerator, Image classification

Authors' addresses: Paolo Meloni, Università di Cagliari, Cagliari, Italy, paolo.meloni@diee.unica.it; Alessandro Capotondi, Università di Bologna, Bologna, Italy, alessandro.capotondi@unibo.it; Gianfranco Deriu, Università di Cagliari, Cagliari, Italy , T3LAB, Bologna, Italy, gianfranco.deriu@unica.it; Michele Brian, T3LAB, Bologna, Italy, michele.brian@t3lab.it; Francesco Conti, Università di Bologna, Bologna, Italy, f.conti@unibo.it, ETH Zurich, Zurich, Switzerland, fconti@iis.ee.ethz.ch; Davide Rossi, Università di Bologna, Bologna, Italy, davide.rossi@unibo.it; Luigi Raffo, Università di Cagliari, Cagliari, Italy, raffo@unica.it; Luca Benini, Università di Bologna, Bologna, Italy, luca.benini@unibo.it, ETH Zurich, Zurich, Switzerland, lbenini@iis.ee.ethz.ch.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2018 Association for Computing Machinery.
1936-7406/2018/1-ART1 $15.00
https://doi.org/0000001.0000001

ACM Reference Format:
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 Trans. Reconfig. Technol. Syst.* 1, 1, Article 1 (January 2018), 24 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

In the last few years, Deep Convolutional Neural Networks have become the go-to solution for most tasks that require human-level understanding of data. Thanks to their outstanding results, they represent the state-of-the-art in image recognition [15, 21, 38], face detection [35], speech recognition [14], text understanding [39, 40] and artificial intelligence in games [24, 43] among other tasks. The big success of CNNs over the last few years can be attributed to the availability of large datasets and to the increasingly large amount of computational power available in General-Purpose Graphic Processing Units (GP-GPUs) to train these networks.

Training of CNNs has been traditionally performed on large servers of General Purpose Processors (GPP) or GP-GPUs since the large variety of algorithms and software frameworks coupled with their high computational complexity require the exploitation of general purpose processors. On the other hand, the regular computational structure of CNN inference, coupled with the inherent parallelism of the convolution operator dominating their computation time, has resulted in a large number of dedicated accelerators much more energy-efficient than general purpose processors [2, 20, 27]. Two notable example of such dedicated accelerators are the Google Tensor Processing Unit (TPU) [20], and the NVIDIA Deep Learning Accelerator (NVDLA) recently released open-source by NVIDIA. Originally designed for the inference task, and given the importance of the learning, Google announced a second, more flexible version supporting floating point operations, also suitable for training of CNNs [13]. High-level tools allow to efficiently implement CNNs on these architectures starting form the CNN model's description created in training frameworks such as PyTorch, TensorFlow or Caffe, abstracting the complexity of the CNN models to the end user.

Embedded architectures for CNN acceleration mainly focus on the inference task, requiring a workload much smaller and regular than training algorithms, and much smaller dynamic and arithmetic precision (e.g. 16-bit fixed point). A widely used category of embedded platforms for CNNs is that of systems-on-chip (SoC) integrating multi-core processors such as ARM Cortex A accelerated with embedded GP-GPUs such as NVIDIA Kepler [28] or Maxwell [29], also featuring LPDDR memory interfaces to sustain the huge memory bandwidth typical of CNNs. Other systems rely on embedded heterogeneous SoCs built around ARM Cortex processors and FPGAs, such as the Xilinx Zynq [41], Xilinx Ultrascale+ [42], and Altera Arria10 [1]. These architectures allow to integrate powerful and efficient accelerators on the reconfigurable logic, exploiting spatial computation typical of application specific integrated circuits (ASIC) rather than thread-level parallelism typical of GP-GPUs. Although high-level synthesis flows allow to implement annotated ANSI-C and OpenCL programs on these heterogeneous systems, plugs to the training environments have been announced by the main FPGA vendors but not made available to developers so far. Several dedicated accelerators have also been proposed in the embedded domain both from companies such as Movidius [26] and from the research community [4, 5, 9]. These architectures are typically implemented as a systolic array of processing elements or more specialized engines focused on the acceleration of convolution-accumulation loops, outperforming all programmable solutions (including FPGAs) in both performance and energy efficiency thanks to the highly optimized implementation approach. However, they all rely on highly specialized architectures and their flexibility is limited due to the lack of general-purpose controlling engines. Moreover, the deployment of these accelerators on real application environments has not been demonstrated, yet.

In this work we propose a CNN accelerator based on the Xilinx Zynq Z-7045 SoC. The proposed accelerator features an operating frequency of 140 MHz resulting into a performance up 169 GOps and an energy efficiency up to 17 GOps/W on end-to-end CNNs. A peculiar feature of the proposed accelerator relies on the presence of one controlling programmable soft-processor on the FPGA which manages the execution of complex CNNs on the Zynq SoC. This approach, which moves the intelligence closer to the compute engine implemented on the FPGA, enables an asynchronous execution model for the proposed accelerator, where the ARM Cortex A9 processor is released from any supervision duty after offloading the commands to the accelerator for the execution of the convolutional layer. This computational paradigm allows to implement a software pipeline where the highly optimized hardware accelerator executes the convolutional layers, while the ARM cores are responsible for the execution of fully-connected layers and data marshaling. Our approach fully leverages the synergy between the A9 cores and the FPGA, heavily exploiting the NEON vector engines to speed up the execution of the software layers, and achieving a very balanced execution time breakdown and very high utilization of all computing resources available on the SoC (hard vector engines and soft FPGA datapath). The accelerator comes with a software environment that allows to automatically generate the ARM host program and the correct memory layout of the weights trained with standard frameworks. The proposed hardware/software architecture is demonstrated through the deployment of the VGG-16 and ResNet-18 CNNs, trained using the Caffe training framework. The evaluated benchmarks achieve a frame rate of 5.5 FPS and 6.6 FPS on the proposed accelerator, respectively, which significantly improves performance and energy efficiency of end-to-end convolutional neural networks over the best-in-class CNN accelerators implemented on the Zynq z-7045 SoC reported in literature. The proposed approach is fully flexible and portable. On the one hand, it allows to easily implement any kind of CNN models fully exploiting the hardware and software capabilities of the Z-7045 SoC; on the other hand, it also eases the porting with big performance benefits to next-generation Ultrascale+ SoC. These SoCs feature a bigger and faster FPGA on the programmable logic (PL), which would allow to host two convolutional engines running at 200 MHz, and they also feature a more powerful processing system (PS) based on a quad-core ARM Cortex A53 processor.

The rest of the paper is organized as follows. Section 2 presents an overview of the state of the art of CNN architectures based on FPGA. Section 3 provides an overview of the computational model of CNNs. Section 4 describes the architecture of the proposed CNN accelerator. Section 4 gives an overview of the software framework that generates the code for the SoC and organize the weights according to the layout required by the accelerator. Section 5 details the implementation of the two CNNs used as use-cases. Section 6 provides a quantitative comparison with the other recently published FPGA CNN accelerators.

2 RELATED WORK

Following the explosion of applications of deep learning algorithms based on CNNs, both academia and industry have focused a significant part of their efforts in the deployment of these algorithms on FPGAs. The hierarchical, relatively simple structure of CNNs, mainly composed of accumulated convolutions with a pre-trained set of filters make them highly suited for FPGA implementation, mainly due to two reasons. First, the large amount of digital signal processing blocks (DSP blocks) enables efficient implementation of the multiply and accumulate elements representing the core of the convolution kernels. Second, as opposed to software programmable solutions such as CPUs and GP-GPUs, the surrounding logic can be adapted to massively exploit the spatial parallelism typical of hardware accelerators, and to customize the local and global memory accesses optimizing them to match the desired computational model.

Several works have tackled the problem of efficiently mapping CNNs onto FPGAs in several application domains which include acceleration of mainstream processors in data-centers, high-end embedded systems running state of the art CNN models, and deeply embedded systems running simpler CNN models that exploit strong quantization of weights to improve performance and energy efficiency at the cost of retraining and classification accuracy. In this section we give an overview of the works that relates more closely with the proposed FPGA accelerator.

Zhang et. al. [44] proposed Caffeine, a hardware/software library to efficiently accelerate CNNs on FPGAs. Caffeine leverages a uniformed convolutional matrix multiplication representation targeting both computation-intensive convolutional layers and communication-intensive fully connected layers of CNN which maximizes the underlying FPGA computing and bandwidth resource utilization. CNN implementations based on Caffeine are implemented with the Xilinx SDAccel high-level synthesis tool integrated in the Caffe learning framework. The implementation of two average-complexity CNN models such as VGG and AlexNet has been evaluated with Caffeine achieving a peak performance of 365 GOps on Xilinx KU060 FPGA and 636 GOps on Virtex7 690t FPGA.

Similarly, Ma et. al. [22] presented an RTL-level CNN compiler that generates automatically customized FPGA hardware for the inference tasks of CNNs from software to FPGA. The approach proposed by [22] relies on a template accelerator architecture described in Verilog including all the main functions employed by CNNs such as convolutions, pooling, etc, which are automatically customized at design time to match the requirements of the target CNN model. This approach allows to exploit the full benefits of low-level RTL design (i.e. frequency, area) while relying on flexible customization which starts from the output of the Caffe learning framework. The proposed methodology is demonstrated with end-to-end FPGA implementations of complex CNN models such as NiN, VGG-16, ResNet-50, and ResNet-152 on two standalone Intel FPGAs, Stratix V and Arria 10, providing average performance up to 720 GOps.

While these two frameworks provide huge performance gains leveraging large FPGA devices such as Virtex7 and Arria 10 FPGAs, they mainly target server applications exploiting batching to improve memory access performance and bandwidth utilization. This approach is not suitable for the embedded applications where cheap and compact SoCs integrating embedded processors and FPGAs are desirable, and images have to be processed in real-time. In this embedded domain, most recent works exploit the capabilities of Xilinx Zynq Z-7045 SoC, integrating a dual-core Cortex A9 processor operating up to 800 MHz and reconfigurable logic featuring 900 DSP slices.

Venieris et. al. [37] presented a latency-driven design methodology for mapping CNNs on FPGAs. As opposed to previous presented approaches mainly intended for bandwidth-driven applications, this work targets real-time applications where the batch size is constrained to one. The proposed design flow employs transformations over a synchronous dataflow modelling framework together with a latency-centric optimization procedure to efficiently explore the design space targeting low-latency designs. This methodology, which relies on Xilinx high-level synthesis tools for mapping (i.e. Vivado HLS) provides extremely high resource utilization (i.e. the totality of the DSP slices of the Xilinx Zynq Z-7045 are employed). However, it has been demonstrated on a relatively simple CNN such as AlexNet, and on a very regular one such as VGG16 featuring only 3×3 kernels, providing a peak performance of 123 GOps. This suggests the current limitations of HLS tools with respect to the template-based approach based on programmable or customizable RTL accelerators proposed in other architectures [22][12][31], including the one proposed in this work.

SnowFlake [12] exploits a hierarchical design composed of multiple compute clusters. Each cluster is composed of four vectorial compute units including a vectorial MAC, vectorial max, a 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). However, although the vector processor-like nature of the accelerator is very flexible, delivering significant performance also for 1×1 kernels, it prevents to fully exploit of spatial computation typical of application specific accelerators, which leads to overheads due to load/store operations necessary to fetch weights and maps from the buffers. This is highlighted by the very poor utilization of the DSP slices available on the FPGA (i.e. only 256 over 900), and by the performance when executing end-to-end convolutional neural networks, which is lower than that of other architectures including the proposed one even though the operating frequency of the CNN engine is significantly higher.

Among CNN FPGA architectures, the precision of arithmetic operands plays a crucial role in energy efficiency. Although most of the architectures available in literature feature a precision of 16-bit (fixed-point)[12, 22, 37] some reduced-precision implementations have been proposed recently, relying on 8-bit, 4-bit accuracy for both maps and weights, exploiting the resiliency of CNNs to quantization and approximation [31].

Qiu et. al. [31] proposed a CNN accelerator implemented on a Xilinx Zynq platform exploiting specific hardware to support 8/4 bit dynamic precision quantization, at the cost of 0.4% loss of classification accuracy. To improve the performance of fully connected layers, mainly limited by the off-chip bandwidth, the architecture employs Single Value Decomposition (SVD) to reduce the memory footprint of the weights. The design was evaluated on a VGG-16 network featuring SVD on the first fully connected layer, and achieves a performance of 187.8 GOps/s and 137.0 GOps/s for CONV layers and full CNN under 150 MHz frequency respectively achieving 4.4 Frames Per Second (FPS).

Most extreme approaches to quantization exploit ternary [30] or binary [36] neural-networks accelerators for FPGA. This approach significantly improves the computational efficiency of FPGA Accelerators, allowing to achieve performance level as big as 8 TOPS [30]. These improvements are due to the 32-bit multipliers that can be replaced by simpler multiplexer and 2's complement operators, while bandwidth for loading weights can be reduced drastically, by 8 to 16 times if we compare with widely used 16-bit fixed point accelerators. The main issue related to binary and ternary accelerator is related to the training. While small networks like MNIST, CIFAR10, SVHN, GTSRB can reach good classification accuracy, the training is still a big challenge for larger networks such as VGG or ResNet [8].

In this work we target execution of state of the art CNNs leveraging 16-bit operands and weights hence not requiring retraining. Starting from the work proposed in [23], we have improved flexibility introducing support for computing kernels different then convolutions. To this aim, we have integrated support for pooling and activation layers and we have implemented and tested tight interaction with the ARM-based processing system in the Zynq, as an instrument to implement end-to-end CNNs.

The peculiarity of the proposed accelerator specifically lies in the execution model: as opposed to all previously published works based on the Z-7045 SoC, where the ARM processors are only responsible for controlling the execution of the CNN, our approach exploit interaction with the processing system (PS) in the Zynq, including the use of the powerful and flexible NEON accelerators, to execute fully connected layers of CNNs. Moreover, our approaches maps on the PS "irregular" computing patterns, that are hard to implement on hardware pipelines. NEURAghe also leverages an asynchronous offload mechanism to enqueue commands to the convolutional accelerators on the programmable logic (PL). This approach allows to implement a software pipeline which overlaps convolutional and fully connected layers fully exploiting the compute capabilities of the Z-7045 SoC significantly improving the performance over best-in-class CNN accelerators implemented on the Zynq z-7045 SoC reported in literature. The proposed approach is highly flexible and portable, and very promising when moving to next generation Zynq Ultrascale+ SoC where the PL is capable

to host two convolutional engines operating at 200 MHz, and the PS is based on a more powerful quad-core ARM Cortex A53 processor.

3 NEURAGHE SYSTEM ARCHITECTURE

3.1 Target computational model

Convolutional Neural Networks can generically be represented as directed graphs in which each edge represents a data tensor, and each node represents an operation (a *layer*) transforming one or more inbound tensors into an outbound tensor. Most often the data tensors considered in CNNs for image processing applications are three-dimensional, with one dimension representing different *channels* or *feature maps* plus two spatial dimensions; especially in the final layers of a CNN topology, some of these tensors can be "collapsed" to 1D vectors, where the spatial notion has been lost. Operations performed in a node can range from convolutions, pooling, and fully connected layers (the most common ones), to generic operations such as tensor concatenation, to special-purpose ones in more exotic cases. Convolutional layers transform a 3D tensor of size $N_\text{i} \times h \times w$ into a new 3D tensor of size $N_\text{o} \times h' \times w'$[1] by means of a combination of convolutions operating on the spatial dimensions followed by a pointwise non-linear activation (often rectification). The linear part of the layer is the following:

$$\text{for } k_\text{o} \in 0 \cdots N_\text{o} - 1, \quad \mathbf{y}(k_\text{o}) = \mathbf{b}(k_\text{o}) + \sum_{k_\text{i}=0}^{N_\text{i}-1} \left(\mathbf{W}(k_\text{o}, k_\text{i}) * \mathbf{x}(k_\text{i}) \right) \tag{1}$$

where $\mathbf{W}$ is the tensor of weights, $\mathbf{b}$ the one of biases, $\mathbf{x}$ is the tensor of input feature maps and $\mathbf{y}$ the one of output feature maps (before activation). Fully connected layers have a similar structure, but they operate on 1D vectors (which can be flattened tensors) and the linear part of the layer is a full matrix-vector multiplication:

$$\mathbf{y} = \mathbf{b} + \mathbf{W} \cdot \mathbf{x} \tag{2}$$

In most CNN topologies, convolutional layers (coupled with pooling) are responsible of the overwhelming majority of operations, and are typically compute-bound due to the high degree of data reuse offered by convolutions; fully connected layers, on the other hand, are responsible for much of the remaining operations, but they are memory-bound due to the absence of reuse. To provide high throughput, a CNN accelerator must therefore be able to speed up the former layers and to hide as much as possible the cost of the latter, which are typically dominated by the memory traffic to fetch the weights. Therefore we designed NEURAGHE taking into account three primary objectives:

(1) support the deployment of arbitrary CNN topologies
(2) acceleration of critical compute-bound operations (i.e. convolutional layers)
(3) hiding of memory-bound operations (i.e. fully connected layers) by overlapping them with the compute-bound ones

To meet these objectives, the NEURAGHE platform employs a hybrid HW-SW scheme in which a *general-purpose processor* (*GPP*) cooperates with a *convolution-specific processor* (*CSP*). The full CNN model is decomposed in the execution of each layer, which can take place either in the GPP or in the CSP, which is dedicated to accelerate the compute-bound convolution tasks and is able to execute also the operations that are more commonly coupled with convolution (activation, padding, pooling).

[1] In most CNN topologies, convolutional layers employ zero-padding of the input to make it so that $h' = h$ and $w' = w$. Convolutions can also use stride greater than 1 in one or both spatial directions.

The CSP and GPP can work concurrently to maximize throughput; however, since most CNN topologies are predominantly sequential, it is sometimes difficult to overlap the execution of convolutional and fully connected layers pertaining to the same execution of the overall model, i.e. to the same input frame. Luckily, in many common CNN topologies such as VGG, fully connected layers are only present at the end of the model. This means that, in presence of a stream of input frames, it is often possible to overlap the execution of convolutional layers pertaining to frame $i + 1$ with that of the final fully connected layers of frame i, effectively hiding the memory-bound operations.

3.2 System architecture

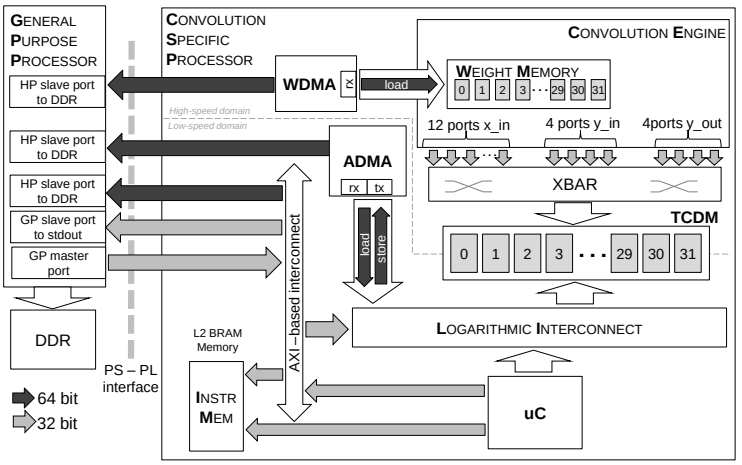

Fig. 1. Architectural template

Figure 1 reports the overall system-level organization of NEURAghe. It is built on top of a Xilinx Zynq SoC and it leverages both the dual Cortex-A9 processing system, which is used as general-purpose processor (*GPP*), and the reconfigurable logic, which hosts the *convolution-specific processor* (*CSP*). NEURAghe exploits two high-performance 64 bit ports for CSP-to-GPP communication (e.g. to access the memory-mapped off-chip DDR) and two general-purpose 32 bit ports for memory-mapped control of the NEURAghe architecture and standard output. As detailed in Section 4, the GPP is used as an active partner in the heterogeneous computation of complex CNN topologies, carrying out tasks that would be accelerated less effectively on the programmable logic, such as memory-bound fully connected layers.

3.3 Convolution-Specific Processor

The Convolution-Specific Processor is composed of several submodules, entirely described in synthesizable SystemVerilog HDL: a local tightly-coupled data memory (*TCDM*) used to store activations and runtime data, a weight memory (*WM*) a weight DMA controller to move weights to the CSP (*WDMA*), an activation DMA to move activations in/out of the CSP (*ADMA*), a simple microcontroller soft-core (μC), and the inner nucleus of the CSP, the actual Convolution Engine (*CE*) that embeds the sum-of-products units used to deploy convolutions on the reconfigurable logic.

The CSP architecture is centered around the local TCDM, which can be concurrently accessed by the uC, the ADMA, a slave port from the GPP, and the CE. The TCDM is implemented with 32 banks of dual-port BRAM primitives, with one port dedicated to direct access from the CE by means of a simple crossbar (*XBAR*), and the other shared between all the other master by means of a low-latency logarithmic interconnect [32] (*LIC*), which arbitrates concurrent access from multiple masters to a single bank by granting only one request using a round-robin starvation free protocol.

The embedded microprocessor is based on a simple OpenRISC core ([10]) coupled with an instruction memory that is accessible on the GPP memory map and is loaded at boot time with a resident runtime environment used to orchestrate the overall CSP operation, e.g. to offload jobs to the CE, program ADMA and WDMA data transfers, notify the GPP of the completion of a CSP job. Both the ADMA and WDMA are based on the DMA described in [33]. With respect to traditional DMA architectures such as the one available in Xilinx Vivado, the DMA employed in this work features a lightweight programming interface with a latency of only 10-15 cycles depending on the kind of transfer, while leveraging on the software programmability of the controlling processor to improve flexibility. Moreover, being tightly coupled to the local memories (WM and TCDM for WDMA and ADMA, respectively) the DMA has been designed without the need of large internal buffers, as opposed to traditional system DMAs that require to store internally the full content of (at least) one AXI burst transaction to reduce the traffic on the system bus. Finally, the DMA has been further extended in this work to support 2D transfers further reducing the programming overhead for this function of general use in CNN computations.

The resident runtime is thoroughly described in Section 4.

The CSP operates on two independent clock domains: the WM, the WDMA, the CE and the XBAR constitute a *high-speed* domain, while the uC, the LIC and the ADMA operate in a *low-speed* one. The dual port banks of which the TCDM is composed are clocked with the two separate clocks according to the connection (high-speed for the CE ports, low-speed for the rest). This allows to maximize throughput for the CE, while keeping full flexibility for the rest of the devices.

3.4 Convolution Engine

The internal architecture of the CE is inspired from the design introduced by Conti et al. [6, 7] as an accelerator of multi-core ultra-low-power system-on-chips. The CE focuses on accelerating convolution-accumulation loops and uses the local TCDM as the source of input feature maps (x) and the storage of output feature maps (y), both fully and partially computed.

As shown in Figure 2, the CE features many connections to the TCDM:

- 12 *x_in* ports, that are used to read input features;
- 4 *y_out* ports, that are used to write partial accumulation results or fully computed output features;
- 4 *y_in* ports, that are used to read previous partial accumulation results.

In each cycle of activity, the CE collects up to 12 input features through *x_in* ports and computes their contributions to 4 output features. The input features *x_in* are loaded through a set of *line buffers*, indicated with *LB* in the diagram, which are used to cache the value of a few lines of the input image so that by loading a single new pixel per cycle an entire new window of the image can be dispatched to the Sum-of-Products (*SoP*) modules to be convoluted with the weight filters. In NEURAghe, the LB blocks are realized by means of shift registers. As the CE works on 16-bit pixel data, each LB can be fed with two pixels per cycle obtained from the input port. After an initial preloading phase, during which the first rows are filled, each LB produces two square convolution windows per cycle, centered on adjacent pixels.

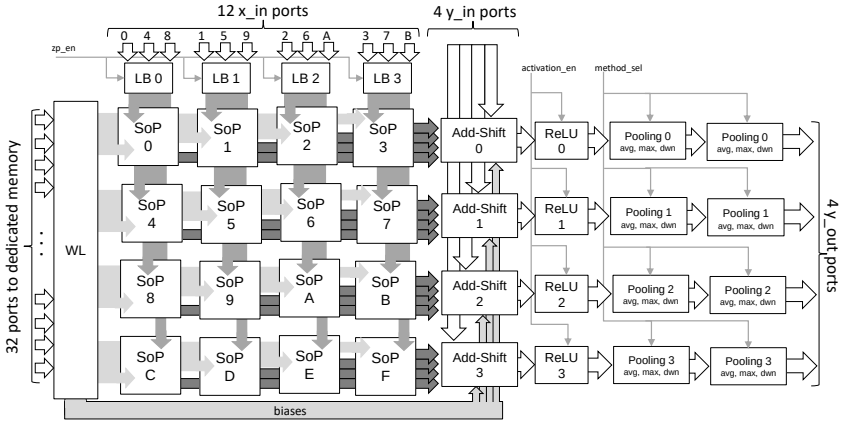

Fig. 2. CE organization

The convolution windows are consumed by the SoP modules, which are the computational core of the accelerator. They apply the bi-dimensional filter kernel to the windows received by the LBs. They are aggressively pipelined with a structure made up of trellises of multiply and add operations (a multiplier, an adder and two pipeline registers, see Section 3.6) to maximize mapping efficiency on the FPGA DSP resources. To cope with the throughput of two convolution windows per cycle produced by the LBs, each SoP module includes two sets of parallel trellises, for a total of $2{\times}N^2$ DSP blocks (where N is the size of the 2D kernel).

Pre-trained weights for a given kernel are loaded in a dedicated register file before the computation starts by a simple weight loader state machine (WL). The WL is directly connected to the private weight memory, composed of a configurable number of BRAM banks and accessible in parallel to minimize weight loading overhead. Referring to the scheme represented in Figure 2, each row of the SoP matrix computes the contributions of input features to the same output feature. Thus, the outputs of the SoP modules in each row must be summed together. Moreover, since output values resulting from multiplication are wider than I/O connections, precision must be adapted to 16 bits with a shift operation, before connection to *y_out* ports. These operations are performed by the *Adder-shifter* module, that is also in charge of the accumulation with previous partial results or with pre-trained bias values.

3.5 Line buffers

In most CNNs, the size of the filtering kernel to be applied may be different for all the convolutional layers. In order to improve the flexibility of our approach, first we have enriched the architecture, integrating line buffers that support different kernel sizes. The configuration proposed in Figure 2, for example, can be reconfigured, by changing the behavior of line buffer modules (please see Fig 3), at runtime by the processing elements in the cluster, to efficiently perform convolutions with 3×3 or 5×5 filters.

In the presented configuration, each SoP modules embeds 27 multipliers for each output pixel (54 in total, since SoP modules produce two output pixels per cycle). The 27 multipliers can be arbitrarily used, depending on the features of the convolution layer to be tackled, to perform either 3 different 3×3 convolutions or one single 5×5 convolution (leaving two multipliers unused in this case).

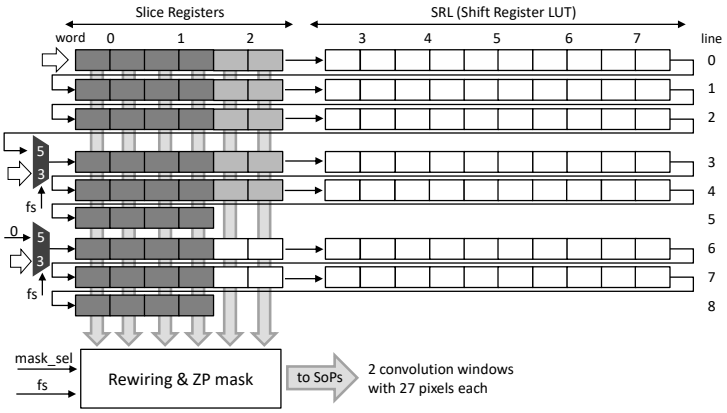

Fig. 3. Reconfigurable line buffer architecture. Gray word slots are accessed only for 5×5 convolution windows, darker word slots are accessed for 3×3 convolution windows. Colored slots in lines from 0 to 4 are used for both configurations. In 5×5 configuration only one stream of input pixels is sent to the first line, while, in 3×3 configuration, the two muxes allow other two input streams to access line buffer from line 3 and 6. The first six words of each line are implemented with register slices, the others words are implemented with Xilinx SRL in order to save resources. Moreover, the content of colored locations are sent to a modules that performs a rewiring to connect slots to the right SoP and to apply zero-padding.

Moreover, to support reconfigurability, the line buffers are capable of switching at runtime between two operating modes, respectively reading one input stream (to be processed with 5×5 filters) or three input streams (to feed the three 3×3 filters computed by each SoP). To this aim, the line buffer is equipped with an additional selection mechanism, controlled via software by means of memory-mapped registers accessible by the cores in the cluster, that can be reconfigured to set the line buffer functionality to the needed operating mode. In the first mode, the line buffer acquires one single stream of pixels and produces in output two windows of 25 pixels each, to be sent to the SoP modules. In the second mode, the shift register is partitioned in three independent regions, used by the line buffer to load three different streams corresponding to three different input features.

In Figure 3, we show the line buffer internal structure, that allows the two mentioned operating modes. As may be noticed, some multiplexers are needed to re-configure the shifting path along the registers in the buffer. Moreover, some rewiring circuitry is needed to select which pixels are part of a convolution window in the considered operation mode and must be forwarded to SoP modules. The buffer locations that correspond to convolution windows in the two modes are highlighted with different colors in the diagram. The same rewiring logic is used to implement zero padding on the input features before convolution, when needed.

The re-configuration of the line buffer takes only one or two cycles and has to be performed at the beginning of the first CE activation in a convolution layer, thus it does not impact on performance.

3.6 SoP modules

SoP modules are implemented using DSP48E1 primitives in the reconfigurable logic of the Zynq device. To achieve high utilization of FPGA resources, different implementation strategies may be chosen. Some works propose *filter size agnostic* approaches. In these design cases [45], multiply-and-accumulate modules are smaller and are reused over different cycles to implement filters of

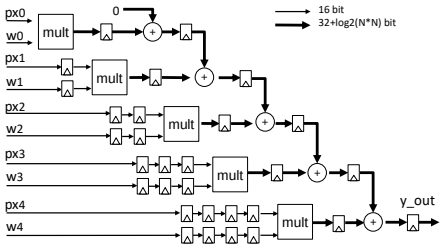

Fig. 4. Single-trellis Sum-of-Products cascade.

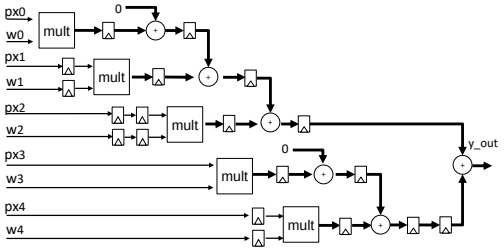

Fig. 5. Multi-trellis Sum-of-Products cascade (with 2 trellises).

different sizes. However, to use most of the FPGA DSP slices at the same time, a higher number of these modules has to be implemented on the device. They are harder to connect with on-chip pixel buffers at the same time, so may be underutilized. Other approaches, on the other hand, implement less hardware filters, using bigger and faster MAC banks, supporting a specific kernel size. In this cases, utilization of the MAC banks is easily higher, at the price of a reduced flexibility. Both kinds of approach show different efficiency when tackling different layer configurations. Our approach aims to be hybrid. We follow a strategy similar to the latter, but we add re-configurability to support most widely adopted filter sizes on the same MAC bank structure, tolerating some underutilization of DSP slices in each bank. The optimal implementation from the point of view of resource utilization would be a single trellis implemented as a cascade of DSP48E1 primitives, that exploits internal multipliers and adders to perform a multiply-and-accumulate operation and the input registers to keep the critical path independent from the size of the considered filtering kernel, as represented in Figure 4. However, in practice this single-trellis SoP couples many DSP48E1 resources tightly together, effectively imposing a restrictive placement constraint in the FPGA place & route phase [2]. This can lead to a reduction of the maximum frequency or too long convergence time in a fairly congested design, in which the target is to use as many DSP48E1 blocks as possible.

To cope with this issue, the SoP structure can also be configured at design time to be partitioned in multi-trellis structures, whose outputs are summed together using a dedicated adder, as shown in Figure 5. Reducing the size of each trellis structure allows for more freedom when selecting the optimal mapping and placement of the resources, improving the overall implementation results and convergence time. In the final NEURAghe design, we used a multi-trellis cascade with 6 trellises.

3.7 Pooling and ReLU module

The CE architecture is also endowed with circuitry implementing computation kernels that may need to be executed right on the output of a convolutional layer in a CNN. Such hardware is placed at the output ports of the CE, as shown in Figure 2, and can be controlled by the host processor using a set of dedicated memory mapped programmable registers. First, the output pixels produced by each port of the convolution engine are streamed into a ReLU (Rectifier Linear Unit) block, that, when enabled, performs rectifier activation function on each pixel. Second, we have integrated on the accelerator a *pooling* layer, that operates on the output streams produced by the convolution engine. This layer is implemented by means of a shift register, that temporarily stores output pixels and compares values of pixels in square pooling windows. After comparison, according to the selected operating mode, the pooling layer outputs one single pixel per window. The pooling layer can be set to perform *max* pooling, *average* pooling or a simple downsampling (statically selecting

[2]DSP48E1 are placed in regular columns in the FPGA fabric in Xilinx Series-7 devices.

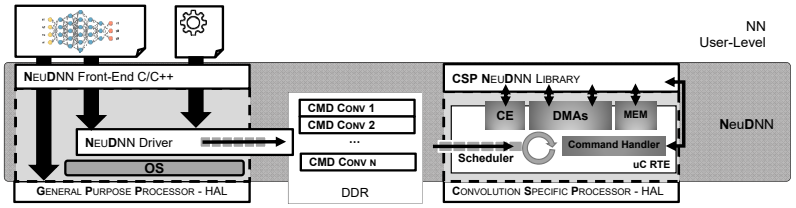

Fig. 6. NeuDNN software stack overview.

the pixel in a specific position in the window). The default configuration of the pooling layer implements pooling over 2×2 windows. Two layers can be cascaded to implement 4×4 windows, alternatively activating at runtime only one or both layers, to dynamically switch between pooling schemes. Different configurations of the module, implementing different basic window sizes, can be chosen at design time.

4 NEUDNN: NEURAGHE DEEP NEURAL NETWORK SOFTWARE STACK

The research field related with neural networks and deep learning represents a hot topic and it is freneticly growing. New layers, new ML tools, and new neural networks topologies are released every day. To tackle this fluid scenario it is crucial to provide a flexible and extensible programming interface that enables the reuse of existing hardware, software and algorithms.

To achieve these objectives we propose a complete and hardware-agnostic software stack, to enable an efficient implementation of Convolutional Neural Networks: the NEURAghe Deep Neural Network software stack (NeuDNN). NeuDNN is an open-source[3] multi-target structured software stack which enables the user to write develop and reuse CNNs to be executed on the presented heterogeneous processing platform. NeuDNN sits on top Linux OS, thus the user is enabled to easily integrate in NN application 3rd Party and legacy software, like JPEG, and OpenCV libs. Figure 6 presents an overview of the whole software stack. It exploits the runtime design proposed Capotondi et al [3] for hereterogenous many-core accelerator and provides a specialized implementation for FPGA-based accelerator.

NeuDNN consists of a C/C++ front-end, which can be used to specify and program CNN at software level, and of a back-end, that maps processing kernels to the hardware accelerator and controls their execution. The back-end – transparent to the user – is composed of a NeuDNN Driver, used to offload computational task to the FPGA-accelerator, and of a Convolution Specific Processor resident RTE, executed by the μC, that receives requests from the driver and schedules autonomously the computation kernels on the Convolutional Engine and data transfers on the DMAs.

To implement a CNN, a user must develop a C/C++ code, exploiting NeuDNN APIs, and must define a simple configuration file describing the target computing platform (for example ARM SoC, or NEURAghe). To load the data needed for the inference, weights and bias values, the user, helped by some migration tools provided by the NeuDNN, can easily import trained models from common ML tools like Tensorflow and Caffe.

4.1 NeuDNN front-end

The NeuDNN Front-End is a configurable C/C++ library for CNN deployment. It gives access to a set of statically linkable functions implementing pre-optimized layers and utilities for CNN

[3]NeuDNN v1.0 will be publically released Q1 2018.

development with no dependency from third party libraries. The NeuDNN targets efficiently ARM class A processors and the NEURAghe architecture, supporting different activation format data types, such as 32-bit IEEE floating point and 16-bit fixed point. Table 1 lists the main computational kernels available as linkable C/C++ API. By default, the library offers optimized implementations for all kernels and the data types deployable to the Generic Purpose Processor (GPP - in this particular case ARM class A cores). All layers are optimized using OpenMP parallel programming model, to exploit parallelisms on the host-side, and ARM NEON vectorization, to exploit SIMD acceleration. When Convolution Specific Processor (CSP) is available, some of those layers can be offloaded to the NEURAghe Convolutional Engine. The CSP-based and the GPP-based implementations share the same APIs, thus the library may forward transparently the execution of the layer to most efficient engine. To enable cooperative computation between the host and the CSP, the hardware accelerated *Convolution** layers support blocking and non-blocking semantics. Like software *tasks*, multiple *Convolution** layers can be enqueued to the accelerator, while the host processor can be used to compute in parallel other layers. These features are enabled by the lower level of NeuDNN software stack.

Kernel	Dimensions	Stride	Data Type	Deploy	Opt.	Note
Convolution	Arbitrary	Arbitrary	float, 16-bit fixed	GPP	OpenMP, NEON	
*Convolution**	1×1, 3×3, 5×5[4]	4,2,1	16-bit fixed	CSP	CE	Async, sync
Max Pooling	Arbitrary	Arbitrary	float, 16-bit fixed	GPP	OpenMP, NEON	
*Max Pooling**	2×2,4×4	4,2,1	16-bit fixed	CSP	CE	After *Convolution**
Avg Pooling	Arbitrary	Arbitrary	float, 16-bit fixed	GPP	OpenMP, NEON	
*Avg Pooling**	2×2,4×4	4,2,1	16-bit fixed	CSP	CE	After *Convolution**
Fully-Connected	Arbitrary	Arbitrary	float, 16-bit fixed	GPP	OpenMP, NEON	
Add	Arbitrary	Arbitrary	float, 16-bit fixed	GPP	OpenMP, NEON	
ReLU	Arbitrary	—	float, 16-bit fixed	GPP	OpenMP, NEON	
*ReLU**	Arbitrary	—	16-bit fixed	CSP	CE	After *Convolution**
Identity	Arbitrary	Arbitrary	float, 16-bit fixed	GPP	OpenMP, NEON	
LRN	Arbitrary	Arbitrary	float, 16-bit fixed	GPP	OpenMP, NEON	

Table 1. Optimized pre-defined NeuDNN Kernels

4.2 NeuDNN Back-End

The NeuDNN back-end is distributed among the GPP and CSP. The GPP side of the back-end is implemented as a driver, in charge of requesting the execution of APIs on the hardware accelerator and of the management of activation/data buffers. The driver takes care of the buffer marshaling and of the general transfers between the host DDR partition and the NEURAghe Convolution Specific Processor. Actions on the accelerator are triggered by the driver by means of dedicated *commands*, consisting in a set of meta-data structures that carry the information needed for the execution of the API (such as weight array pointers, activation array pointers, etc.). Commands are stored in a shared FIFO queue mapped on the DDR address space. Being NeuDNN implemented on top of the Linux OS, the DDR must be split in two partitions: one used by the OS as main virtual memory; and other one, unmapped and accessed by `/dev/mem`, contiguous and not paged, used to share data buffers between GPP and CSP.

The CSP-side is fully managed by a resident runtime, executed by the μC in the CSP, which is loaded and activated at the startup of the system, just after the load of the bitstream on the programmable logic. The runtime, written in C, has direct access to the CSP HAL and is in charge of orchestrating data transfers from/to the local Convolutional Engine TCDM and triggers of CE activations. The runtime decomposes commands received by the GPP driver, requesting CNN basic

[3]Bigger convolutional filters can be implemented using a composition of these dimentions (see Section 5)

operations such as Convolutions, Max Pool layers and ReLUs, into a scheduled track of elementary operations on the CE and on the two DMAs. The used scheduling strategy is aggressively optimized to improve efficiency under limited bandwidth availability, using double-buffering and sliding window techniques to optimize the overlapping of computation with data transfers. A pseudo-code generally describing such strategy is presented in Figure 7.

```
while (TRUE)
      wait_for_cmd()
      nb_y_groups = num_of / of_in_parallel
      nb_x_groups = num_if / if_in_parallel
      for o in range(0, nb_y_groups):
          wait_for_stores()
          for i in range(0, nb_x_groups):
              load( x_loc[i%2] )
              load( w_loc[i%2] )
              if o>0:
                  store( y_loc[(o-1)%2] )
              wait_for_loads()
              conv(x_loc[i%2], w_loc[i%2], y_loc[o%2])
      store( y_loc[o%2] )
```

Fig. 7. Pseudo-code representing the scheduling strategy executed by the uC. The accelerator consumes up to 12 input features and produces up to 4 output features in parallel so the features maps are grouped in a number of y_groups and a number of x_groups. For each new offload two nested loops are performed. There are two memory areas - one for inputs and one for outputs - both implementing double buffering. Each iteration of the inner loop switches the input buffer, starts the transfers and triggers the convolution and each iteration of the outer loop switches the output buffer. The store transfers are programmed only after the first convolution and a store if performed after the last convolution outside the loops. The uC must wait if a buffer slot is non ready to start the convolution or to start the transfers. After the first inner loop transfers and convolutions are overlapped.

5 EXPERIMENTAL RESULTS

To evaluate the performance of NEURAghe and the flexibility of NeuDNN on real-world CNN topologies, we used our framework to implement two of the most commonly used ones: VGG-16 [34] and ResNet-18 [16]. These two networks enable to show different computational approaches that can be supported using our framework, like computational pipelining and cooperative computation between the General Purpose Processor and the Convolution Specific Processor. While the first one exploits the synergistic use of computing resources to take profit of parallelism and to improve performance, the second one maps on the ARM-based processing subsystem all the computational tasks that are hard to accelerate on highly-parallel hardware structures, like the SoP modules inside CE, such as data marshaling and shortcut implementation. Moreover, as an integration to the second use-case, we present an experiment related with the acceleration of a light-weight CNN topology, to provide an insight on the possibility of accelerating with NEURAghe recent algorithms conceived for extensive workload reduction. The results show up to 225 GOps/s[5] delivered by the Convolution Specific Processor, and an end-to-end classification frame-rate on ImageNet up to 6.6 fps on ResNet-18, and 5.5 fps on VGG-16.

[5] As is often the case for CNNs, we count a multiply-accumulate as two operations. Using the notation of Section 3.1 ($K_{h,w}$ denote the height and width of 2D filters), the performance of a convolutional layer is given by

$$\text{Perf[GOps/s]} = 2 \times N_i \times N_o \times h' \times w' \times K_h \times K_w \;/\; t_{\text{execution}}$$

As discussed in Section 3, NEURAghe is deployed on a Xilinx Zynq Z-7045 SoC. The two ARM Cortex A9 in the GPP are clocked at 800MHz, while the CSP operates at 70MHz in the low-speed domain and at 140MHz in the high-speed one, including the CE. In this configuration, the GPP OS uses 744MB of the Xilinx PS DDR3, while the rest of the DDR3 (256MB) is used as contiguous shared memory accessible by both the GPP and the CSP. The GPP is equipped with a Ubuntu 16.06 LTS OS (Linux Kernel 3.8) and the toolchain used for compilation was GNU GCC v5.4.

5.1 Hardware implementation evaluation

In the presented configuration, the 16 SoP modules, including 54 DSPs each to produce two output pixels per cycle each, are clocked at 140 MHz. The configuration features four reconfigurable line buffers, each capable of loading up to 128 words (256 pixels). This means that the proposed configuration can process input features which are up to 256 pixel wide. This size is adequate for most of state-of-the art CNN benchmarks. Processing of wider input features requires their partitioning in sub-stripes, by means of dedicated software routines.

Table 2 shows the FPGA resource utilization of the proposed architecture, when mapped on the Zynq Z-7045.

	DSP	BRAM	LUTs (logic)	LUTs (SR)	Regs
Used	864	320	88154	11397	61250
Avail.	900	545	218600	218600	437200
%	96.0%	58.7%	35.1%	16.2%	14.1%

Table 2. Resource utilization on Zynq Z-7045

	DSP	BRAM	LUTs (logic)	LUTs (SR)	Regs
Used	1728	640	146573	22385	114261
Avail.	2520	912	274080	144000	548160
%	68.6%	70.2%	53.5%	15.6%	20.8%

Table 3. Resource utilization on Zynq UltraScale

As may be noticed, the mapping uses 864 out of the 900 DSP blocks available in the device. Thus the proposed configuration uses almost all of the processing power available in the device. BRAM utilization is around 35%, thus L2 and TCDM size can be comfortably increased if required by the use-case. Also utilization of LUT and registers is low. There is a significant number of LUTs used as shift-registers, due to the internal organization of the line buffer. All the buffer segments that do not need to adapt to different uses and have a static shift path, have been described in HDL to infer use of LUTs, to obtain a faster and less resource-hungry implementation. It is worth highlighting that the CSP uses only two of the 4 HP ports connecting the programmable logic to the PS and the DDR3. This means that our approach can be scaled easily replicating the number of CSPs in a bigger devices. According to our scaling experiments, performed with a Vivado synthesis, a Zynq UltraScale XCZU9EG-2FFVB1156 FPGA would be able to host two CSPs, both clocked at 200 MHz and able to independently access the PS to communicate with the DDR3 memory.

5.2 VGG-16

VGG is a deep convolutional neural network proposed by K. Simonyan & A. Zisserman [34]. The model achieves up to 92.7% top-5 test accuracy in ImageNet classification [18]. Figure 8 shows the structure of VGG-16. It consists of five computational *blocks* followed by three fully-connected layers. Each computational *block* is composed of two or three 3×3 convolutional layers followed by a max pooling reduction.

Compared to the standard VGG-16 network proposed by K. Simonyan & A. Zisserman, in this work we exploited the SVD compression methodology proposed by Girschik et al. [11, 31] for the first fully-connected layer (FC6). This compression enables to reduce the memory footprint and the computational complexity of the FC6 layer of 3×, with an accuracy loss smaller than 0.05%.

VGG-16 NEURAghe deployment. Mapping VGG-16 on NEURAghe is straightforward. The five computational blocks can be enqueued to the CSP without any interaction with the GPP, while

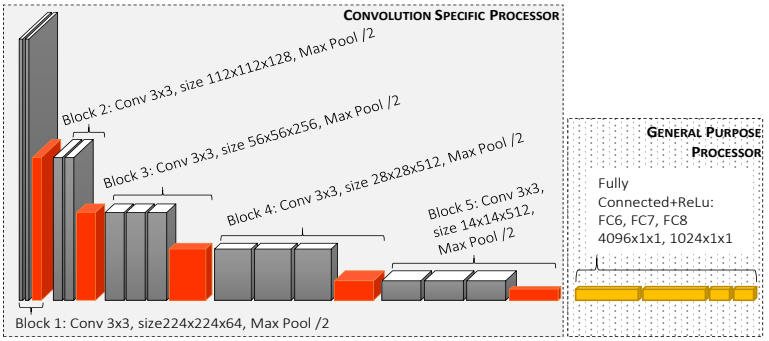

Fig. 8. VGG16 topology[34]. Gray layers indicate 3×3 convolutional layers; the Max Pooling reduction layers are in red, while the Fully Connected layers are in yellow. The two surrounding boxes highlight how the layers are mapped on the NEURAghe platform.

the fully connected layers can be fully executed on the GPP. Compared to the original model, the NEURAghe implementation of VGG-16 requires two additional layers to manage the *data marshaling* from/to the CSP - the first such operation is performed before the first VGG-16 block and the second between the last computational block and the first fully-connected layer. The *data marshaling* - as discussed in section 4 - consists in the transfer of data from/to the OS-managed DDR section and the shared contiguous memory DDR partition, and the inter/deinter-lacing of activations. The VGG-16 implementation uses 16-bit fixed-point data quantization for activations, weights, and bias, using Q5.11 format.

Table 4 resumes activation size, measured execution time, and performance in GOps/s for all VGG-16 components (with the exception of data marshaling layer), divided in the respective computational blocks. From the profiling, we can first observe that the total data marshaling overhead is below 13ms, i.e. less than 5% of the whole latency. Together, all the VGG-16 computational blocks take 181ms, providing an average throughput of 169.7 GOps/s. With the exception of the first convolutional kernel – which offers a limited number of input features and then a limited possibility of parallelism for the Convolutional Engine – the other convolutional kernels generate more than 100 GOps/s, with a peak performance of 225 GOps/s. The fully-connected layers require on the 70 ms, with an average performance of 1.02 GOps/s. As we previously discussed, these layers are strongly dominated by the memory bandwidth. The overall measured latency is 263.61 ms with a global average performance of 122.58 GOps/s.

Thanks to the high flexibility of our proposed architecture and software stack, different execution models can be implemented to extract better performance. Considering the common scenario where the input images are frames from a video stream, we can take advantage of the strong segregation of layers between the Convolution Specific Processor and the General Purpose Processor to improve the overall throughput of the VGG-16 applying a three-stage pipeline. This is implemented by enqueuing the execution of convolutional blocks in asynchronous fashion, and letting the GPP execute the fully connected layers for frame $i-1$, while the convolutional blocks of frame i are being computed by the CSP. A third stage is added to remove the overhead of the first data marshaling from the critical path of the SW pipeline.

VGG-16 performance analysis. The VGG16 is then split in three stages as follow:

- **Stage I:** consists only of the the first data marshaling layer.

[6] Average GOps/s on convolutional layers.

	Size	Time (ms)	GOps/s
Marshaling	3×224×224	9.804	—
Block 1	64×224×224	13.999	12.38
	64×224×224	32.784	112.8
Block 2	128×112×112	10.417	177.54
	128×112×112	18.14	203.92
Block 3	256×56×56	8.901	207.76
	256×56×56	17.132	215.9
	256×56×56	16.833	219.76
Block 4	512×28×28	8.68	213.04
	512×28×28	16.578	223.12
	512×28×28	16.428	225.12
Block 5	512×14×14	7.093	130.36
	512×14×14	7.092	130.36
	512×14×14	7.128	129.68
Marshaling	512×7×7	2.9	—
FC 6-SVD	4096×1×1	32.554	0.917
FC 7	4096×1×1	29.46	1.138
FC 8	1000×1×1	7.688	1.065
Latency		**263.61**	**122.58 (169.34[6])**
Pipelined		**181.205**	**169.74**

Table 4. VGG-16 measured performance on NEURAGHE

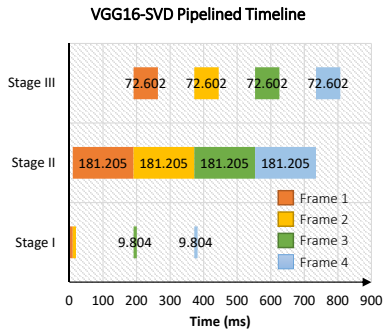

Fig. 9. VGG16 four frame execution timeline

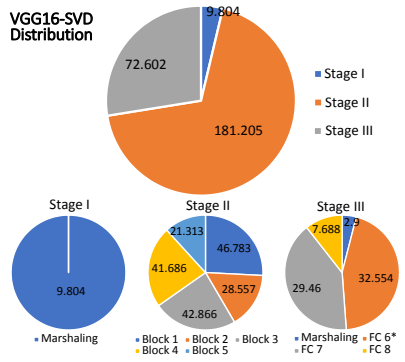

Fig. 10. VGG16 pipeline time distribution in ms

- **Stage II:** consists of all the computational blocks executed on the Convolution Specific Processor.
- **Stage III:** consists of all the rest of layers (marshaling, and fully-connected) executed on the General Purpose Processor.

A clear view of the execution in pipeline of VGG-16 is given by the Figure 9. The figure shows a real timeline, profiled on a NEURAGHE board, of the execution of VGG-16 on 4 frames. Figure 10 shows how the execution time are distributed among the stages. Pipelined execution, thanks to the heterogeneous cooperative computation between GPP and CSP, enables to drop per-frame execution time of VGG-16 to 181.2 ms, corresponding to an average throughput of 169.74 GOps/s.

5.3 ResNet-18

Very deep neural networks are often difficult to train; the class of Residual Deep Neural Networks aims to solve this issue by providing "shortcut" paths between the first and the last layers, improving their correlation at training time, at the cost of a more complex and less regular topology. ResNet-18 [16] is one of the first representatives of this class of topologies, which won the 1st place on the

ILSVRC 2015 classification task. These kind of networks are more and more common as they are typically smaller and have lower memory footprint than simpler topologies of equivalent accuracy.

ResNets are built upon a simple basic block consisting in the sum of the results of a chain of several convolutional layers applied on an activation tensor $\mathbf{x}$ with a "shortcut" to $\mathbf{x}$ itself, sometimes augmented by a 1×1 convolution layer. Due to their more complex topology, ResNets are less straightforward to deploy on hardware, however the NeuDNN software stack is able to fully manage this kind of topology.

ResNet-18 NEURAghe deployment. Figure 11 shows the ResNet-18 topology. The left graph shows the original ResNet-18 neural network as proposed by He K. et al. [16] side-by-side with the optimized implementation for NEURAghe. In black we highlighted the layers that can be forwarded to the Convolution Specific Processor, while the grey boxes are layers that can be executed only on the General Purpose Processor.

In this case, three main modifications were applied to extend the usage of the Convolution Specific Processor. First, the 7×7 convolutional layer, which is not natively supported by the Convolutional Engine, was split in four 5×5 convolutional layers followed by a software managed merge layer. Second, the batch normalization layers, which at inference time simply apply a static pointwise linear operation, where merged with convolution layers by embedding the scaling and translation factors within the convolution weights and biases, respectively [19]; ReLU activations are also performed by the Convolution Engine. Third, the 1×1 convolutions (not natively supported by the Convolution Engine) are mapped on 3×3 layers.

Similarly to VGG-16, data marshaling layers were added between computations run on CSP and GPP when necessary. For pointwise operations (e.g. the shortcut merge operations composed of a sum and a ReLu, which runs on the GPP) the interlacing of activations is irrelevant, and thus data marshaling operations around them can be safely skipped. This is obviously not true for max pooling and fully connected layers.

Like VGG-16, our ResNet-18 implementation use 16-bit fixed point arithmetic for activations, weights, and bias, with Q5.11 format.

ResNet-18 performance analysis. Figure 12 plots the execution time measured in milliseconds on the NEURAghe platform.

The most time-consuming blocks are the four marshaling layers due to the split of the 7×7 convolution in four smaller ones. Each marshaling action takes up to 14 ms, mainly due to the fact that the amount of data to move and process is significant (64×112×112 pixels). The second most time consuming layer is the merging of partial results for the emulated 7×7 convolutions, and the max pooling that is in a configuration not supported on the accelerator (3×3 with stride 2). Both layers take around 9 ms. 5×5 convolutions take ~4 ms, and are penalized by limited number of input activations and the stride 2. However, thanks to the asynchronous offloading of convolutions to the CSP, these overheads can be partially overlapped with the execution on the accelerator, and can be also parallelized among the two ARM Cortex A9 due to the independence of data marshaling stages with one another. Thus, while the sum of all the execution time of the layers used to emulate the 7×7 convolution is 92.0 ms, the end-to-end execution time measured is only 51.2 ms, showing up to 40 ms gain due to the cooperative computation of the GPP and the CSP. The last convolutions are penalized as well, in this case due to the small input feature maps size (only 7×7 pixels) which causes a sub-utilization of the hardware resources in the CE. Considering the overlaps, the measured end-to-end execution time for the whole neural network inference is 150 ms, equivalent to a frame rate of 6.6 fps.

Figure 13 shows the time distribution of each component. The convolutions take around 48% of the whole time, while 42% is spent on data-marshaling – most of it due to the 7×7 convolution.

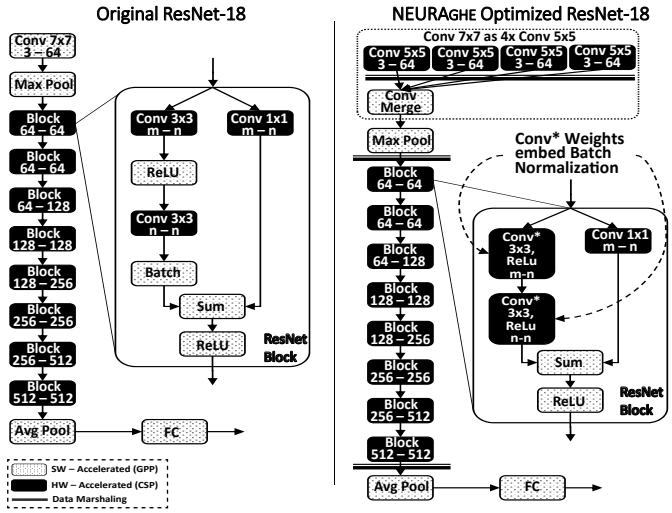

Fig. 11. ResNet-18 topologies. Left topology is the original ResNet-18 as proposed by [16], while to the right the optimized implementation for NEURAGHE

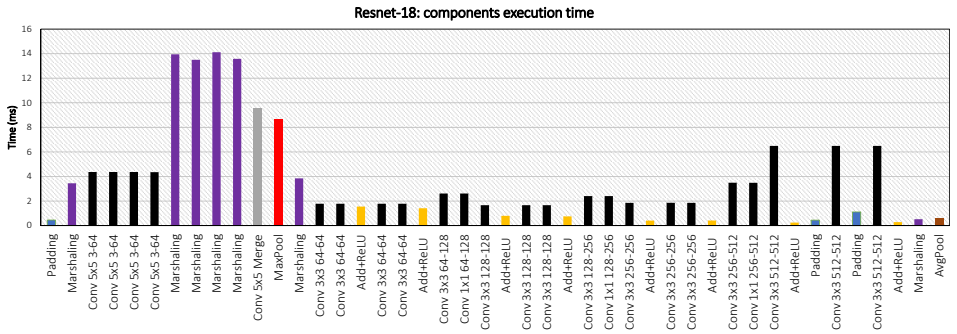

Fig. 12. ResNet-18 layer-by-layer profiling.

While the emulated version is not particularly efficient, a pure software execution on the GPP would take up to 176 ms (0.6MOps/s) – far away from the performance achieved even in a sub-optimal operational region of the CSP.

Finally, Figure 14 shows the measured GOps/s for all the convolutional layers. For ResNet-18, NEURAGHE provides up to 140 GOps/s at peak. On average, throughput drops to 58.4 GOps/s due to two main reason: the striding in output of some of the convolutions, and the 1×1 convolutions. This is because in layers with stride values higher than 1, performance is limited by the line buffer functionality. It keeps loading two pixel per cycle from each port but some convolution windows must be discarded, causing idle cycles in the accelerators. 1×1 convolutions are also sub-optimal since a SoP module is under-utilized to perform only 2 MAC operations per cycle, lowering the performance level of the CE.

Other "irregular" topologies: SqueezeNet light-weight CNN. Newly appearing topologies aim at reducing by design the overall workload related with inference, to be easily mapped on ultra-low

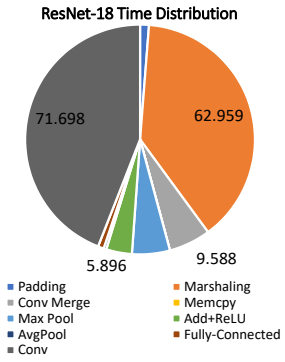

Fig. 13. ResNet-18 execution time distribution (milliseconds)

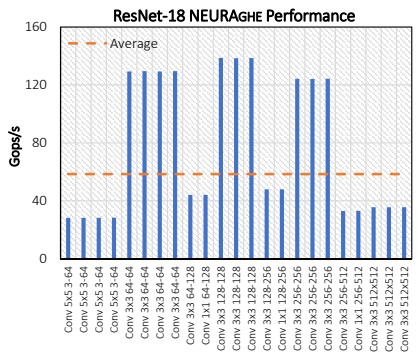

Fig. 14. ResNet-18 Convolutional Engine throughput

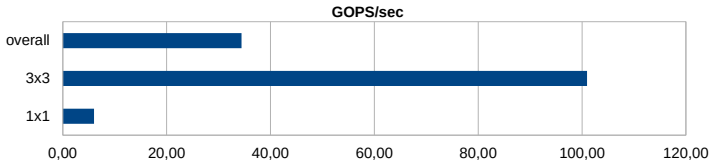

Fig. 15. SqueezeNet convolutional layer profiling.

power platforms and IoT nodes. Thus, such CNN schemes are not the main target of our IP. In this work, NEURAGHE has been configured and synthesized to exploit as much as possible a mid-to-high end Zynq device, considering as tasks more widely diffused and known compute-intensive CNNs. Nevertheless, to assess NEURAGHE on this class of light-weight algorithms, we have evaluated the performance level that may be achieved on the SqueezeNet topology [17]. NEURAGHE is capable of achieving a throughput corresponding to almost 15 fps and to around 30 GOPS/s. As may be noticed in Figure 15, performance bottleneck are 1×1 layers, that have been extensively used in SqueezeNet to reduce the overall computing complexity. Nevertheless, using NEURAGHE we could exploit heterogeneous computing to dedicate the GPP to data marshaling and feature merging, still providing performance aligned with what reported in literature [25]. However it is worth pointing out that, for similar use cases, a smaller configuration of NEURAGHE, easily implementable on smaller devices simply reconfiguring CSP parameters, would most likely be a better match.

5.4 GPP-accelerated layers performance analysis

As we discussed, NeuDNN is able not only to exploit the CSP, but also to accelerate other layers that traditionally do not allow optimal mapping on the programmable logic, by means of the capabilities of the ARM Cortex-A9 core. This is based on two well known methodologies: *thread-level* parallelization, which can be accessed by means of the OpenMP programming model, and *SIMD vectorization*, which is enabled by the NEON vector unit featured by each ARM core, supporting a combined 64- and 128-bit SIMD instruction set for media and signal processing applications.

To measure the effectiveness of our implementations, we analyzed the performance generated by the NeuDNN layers executed on the GPP for VGG-16 and ResNet-18 using the well known *roofline model* (Figure 16). The two ARM Cortex-A9, running at 800MHz, are able to deliver up to 6.4 GFlop/s, and the main memory is limited to 4GB/s. The *computational density* threshold between

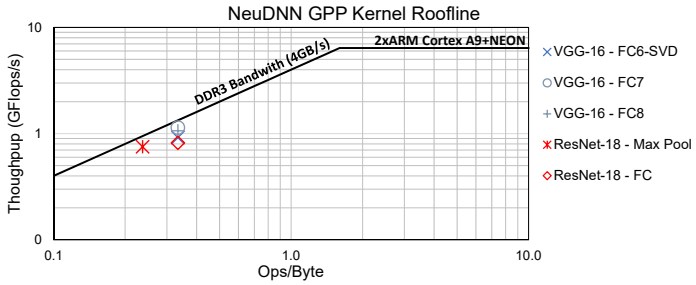

Fig. 16. Roofline Model of NeuDNN layers for Xilinx Zynq-7045 SoC

memory-bound and *compute-bound* operation is in this SoC around 1.5 Op/B. As recalled in Section 3.1, most non-convolutional layers, in particular fully connected layers, are heavily memory bound: each weight is used only once. This is confirmed in the case of our two target CNNs: we measured a computational density of 0.2-0.3 Op/B, which is well within the memory-bound region. As can be seen in Figure 16, the NeuDNN software-accelerated implementations are essentially hitting the performance roof set by the memory bandwidth and are therefore optimal given the underlying Zynq architecture.

5.5 Comparison with State of The Art

To better understand how the proposed architecture performs with respect to other FPGA accelerators in the state-of-the-art, Table 5 provides a comparison with a subset of competitive accelerators dedicated to embedded CNN inference, and deployed on the same Xilinx z-7045 board. For this reason, all the accelerators show a similar power consumption of 9-10W. Apart from this, significant differences exist between the various platforms.

In terms of raw performance, NEURAghe demonstrates 18-27% better results than the competing platforms on VGG-16, which is often used as a performance benchmark. The accelerator proposed by Vernieris et al. [37] and Snowflake [12] claim a performance up to 123 GOps/s and 122 GOps/s, respectively, which is 27% smaller than the performance of NEURAghe, and 18% smaller than the performance of the accelerator proposed by Qiu et al. [31]. In the case of Vernieris et al., performance is mainly limited by the lower operating frequency, which might be attributed to the high-level synthesis methodology, which is not always guaranteed to reach optimal results in terms of implementation. For what concerns SnowFlake, their operating frequency is the highest, but they use the lowest amount of DSP resources, which negatively impacts their peak performance. Although they claim that their performance should be scalable by replicating the accelerator design on the same device, a higher occupation of the PL might result in a more congested - and therefore lower frequency - design. While they report results for ResNet-50, a CNN sharing a similar topology with ResNet-18, it is impossible to perform a direct comparison with their result, as contrarily to the other works they do not report end-to-end performance, but take into account only convolutional layers. Qiu et al. is the strongest competitor to our work, as they deliver up to 138 GOps/s on VGG-16 – ~18% less than NEURAghe. The critical advantage provided by our work is that NEURAghe fully exploits both the programmable logic and the GPP, "embracing" a heterogeneous programming model. This allows us *i*) to overlap the execution of the fully connected layers and the convolutional layers, and *ii*) to use the NEON extensions on the dual-core ARM Cortex-A9.

	NEURAghe	Qiu et al. [31]	Gokhale et al. (Snowflake) [12]	Venieris & Bouganis [37]
Platform	Xilinx Zynq Z-7045	Xilinx Zynq Z-7045	Xilinx Zynq Z-7045	Xilinx Zynq Z-7045
Clock (MHz)	140MHz	150MHz	250MHz	125MHz
Power (W)	~10W	9.63W	9.61W	~10W
DSP	864	780	256	900
LUT	100K	183K	—	—
FF	61K	128K	—	—
BRAM	320	486	—	—
Actual Perf. (GOps/s)	169 (VGG-16) 58 (ResNet-18)	138 (VGG-16) —	122[1](ResNet-50) 120[1](AlexNet)	123 (VGG-16) —
Frame/s	5.5 (VGG-16) 6.6 (ResNet-18)	4.46 (VGG-16) —	17.7[1](ResNet-50) 100.3[1](AlexNet)	4.0 (VGG-16) —
End-2-End	yes	yes	no	yes
Quantization	16 bit fixed	16/8/4 bit fixed	16 bit fixed	16 bit fixed

[1] Does not include the final fully connected layers.

Table 5. NEURAghe Performance Summary and System Comparison

6 CONCLUSION

We have presented NEURAghe, a Zynq-based processing platform for CNN, specifically designed to improve flexibility and re-usability in different context and for the implementation of different CNN-based algorithms. Our approach relies on the tight interaction between software and hardware. The ARM processing system in the Zynq is not only used for housekeeping tasks, but is also used at its best to perform computation tasks when needed. Moreover, the accelerator implemented in the programmable logic is also controllable via software, integrating a microcontroller in charge of finely managing the basic operations of the other building blocks. We also developed a complete software stack, acting as a distributed runtime on the processing system and on the microcontroller to ease the life of users willing to implement a new CNN case on NEURAghe.

We have shown with two different experiments on NEURAghe that an approach based on heterogeneous processing, simultaneously exploiting programmable logic and ARM-based processing system, can be used effectively for different reasons. In a first CNN, VGG-16, we have shown that it can be used to improve performance, performing 18% better than the best competitor in literature. Under the workload imposed by ResNet-18, we have shown that it can be used with success to improve flexibility, implementing on the processing system "irregular" CNN kernels and "adaptation" layers not supported by the accelerator. Our approach is highly-reusable, relying on a completely sw-programmable stack, and scalable, we have successfully implemented two clusters on a Ultrascale+ device, clocked at 200 MHz. Thus, it paves the way for the exploitation of a new acceleration paradigm, relying on hardware-software tight synergy, in the upcoming future of CNN development. It will be a key technique to face challenges posed by next generation of newly appearing CNN algorithms, increasingly irregular and complex, using next-generation of All-Programmable SoCs, increasingly powerful and heterogeneous.

ACKNOWLEDGMENTS

This project has received funding from the European Union's HORIZON 2020 Research and Innovation programme under grant agreement No. 780788 (https://www.aloha-h2020.eu/) and No. 732631 (http://oprecomp.eu/).

REFERENCES

[1] Altera. 2017. *Altera Arria 10.* https://www.altera.com/products/fpga/arria-series/arria-10/overview.html.

[2] E. Azarkhish, D. Rossi, I. Loi, and L. Benini. 2017. Neurostream: Scalable and Energy Efficient Deep Learning with Smart Memory Cubes. *IEEE Transactions on Parallel and Distributed Systems* PP, 99 (2017), 1–1. https://doi.org/10.1109/TPDS.2017.2752706

[3] A. Capotondi, A. Marongiu, and L. Benini. 2017. Runtime Support for Multiple Offload-Based Programming Models on Clustered Manycore Accelerators. *IEEE Transactions on Emerging Topics in Computing* (2017).

[4] L. Cavigelli and L. Benini. 2016. A 803 GOp/s/W Convolutional Network Accelerator. *IEEE Transactions on Circuits and Systems for Video Technology* PP, 99 (2016), 1–1. https://doi.org/10.1109/TCSVT.2016.2592330

[5] Y. H. Chen, J. Emer, and V. Sze. 2016. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 367–379. https://doi.org/10.1109/ISCA.2016.40

[6] F. Conti and L. Benini. 2015. A Ultra-low-energy Convolution Engine for Fast Brain-inspired Vision in Multicore Clusters. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE '15)*. EDA Consortium, San Jose, CA, USA, 683–688.

[7] F. Conti, R. Schilling, P. D. Schiavone, A. Pullini, D. Rossi, F. K. Gürkaynak, M. Muehlberghuber, M. Gautschi, I. Loi, G. Haugou, S. Mangard, and L. Benini. 2017. An IoT Endpoint System-on-Chip for Secure and Energy-Efficient Near-Sensor Analytics. *IEEE Transactions on Circuits and Systems I: Regular Papers* 64, 9 (Sept 2017), 2481–2494. https://doi.org/10.1109/TCSI.2017.2698019

[8] M. Courbariaux, Y. Bengio, and J. David. 2015. BinaryConnect: Training Deep Neural Networks with binary weights during propagations. In *Advances in Neural Information Processing Systems*. 3105–3113. arXiv:1511.00363

[9] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam. 2015. ShiDianNao: Shifting vision processing closer to the sensor. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 92–104. https://doi.org/10.1145/2749469.2750389

[10] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini. 2017. Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25, 10 (Oct 2017), 2700–2713. https://doi.org/10.1109/TVLSI.2017.2654506

[11] R. Girshick. 2015. Fast R-CNN. In *International Conference on Computer Vision (ICCV)*.

[12] V. Gokhale, A. Zaidy, A. X. M. Chang, and E. Culurciello. 2017. Snowflake: An efficient hardware accelerator for convolutional neural networks. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1–4. https://doi.org/10.1109/ISCAS.2017.8050809

[13] Google. 2017. *Build and train machine learning models on our new Google Cloud TPUs.* https://www.blog.google/topics/google-cloud/google-cloud-offer-tpus-machine-learning/.

[14] A. Hannun et al. 2014. Deep Speech: Scaling up end-to-end speech recognition. *Computing Research Repository* abs/1412.5567 (2014).

[15] K. He, X. Zhang, S. Ren, and J. Sun. 2015. Deep Residual Learning for Image Recognition. *ArXiv:1512.03385* (Dec. 2015). arXiv:cs.CV/1512.03385

[16] K. He, X. Zhang, S. Ren, and J. Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[17] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *CoRR* abs/1602.07360 (2016). http://dblp.uni-trier.de/db/journals/corr/corr1602.html#IandolaMAHDK16

[18] Image-Net. 2017. *Large Scale Visual Recognition Challenge.* http://image-net.org/.

[19] S. Ioffe and C. Szegedy. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *CoRR* abs/1502.03167 (2015). arXiv:1502.03167 http://arxiv.org/abs/1502.03167

[20] N. Jouppi et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/3079856.3080246

[21] A. Krizhevsky, I. Sutskever, and G. Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[22] Y. Ma, Y. Cao, S. Vrudhula, and J. s. Seo. 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)*. 1–8. https://doi.org/10.23919/FPL.2017.8056824

[23] P. Meloni, G. Deriu, F. Conti, I. Loi, L. Raffo, and L. Benini. 2016. Curbing the Roofline: A Scalable and Flexible Architecture for CNNs on FPGA. In *Proceedings of the ACM International Conference on Computing Frontiers (CF '16)*. ACM, New York, NY, USA, 376–383. https://doi.org/10.1145/2903150.2911715

[24] V. Mnih et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (26 Feb 2015), 529–533. Letter.

[25] P. G. Mousouliotis and L. P. Petrou. 2018. SqueezeJet: High-level Synthesis Accelerator Design for Deep Convolutional Neural Networks. *ArXiv e-prints* (May 2018). arXiv:cs.CV/1805.08695

[26] Movidius. 2017. *Movidius Neural Compute Stick: Accelerate deep learning development at the edge.* https://developer.movidius.com/.

[27] NVIDIA. 2017. *NVIDIA Deep Learning Accelerator (NVDLA).* http://nvdla.org/.

[28] NVIDIA. 2017. *NVIDIA Tegra K1.* http://www.nvidia.com/object/tegra-k1-processor.html.

[29] NVIDIA. 2017. *NVIDIA Tegra X1.* http://www.nvidia.com/object/tegra-x1-processor.html.

[30] A. Prost-Boucle, A. Bourge, F. Petrot, H. Alemdar, N. Caldwell, and V. Leroy. 2017. Scalable high-performance architecture for convolutional ternary neural networks on FPGA. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL).* 1–7. https://doi.org/10.23919/FPL.2017.8056850

[31] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang. 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).* ACM, New York, NY, USA, 26–35. https://doi.org/10.1145/2847263.2847265

[32] A. Rahimi, I. Loi, M. R. Kakoee, and L. Benini. 2011. A fully-synthesizable single-cycle interconnection network for Shared-L1 processor clusters. In *2011 Design, Automation Test in Europe.* 1–6. https://doi.org/10.1109/DATE.2011.5763085

[33] Davide Rossi, Igor Loi, Germain Haugou, and Luca Benini. 2014. Ultra-low-latency Lightweight DMA for Tightly Coupled Multi-core Clusters. In *Proceedings of the 11th ACM Conference on Computing Frontiers (CF '14).* ACM, New York, NY, USA, Article 15, 10 pages. https://doi.org/10.1145/2597917.2597922

[34] K. Simonyan and A. Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[35] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf. 2014. DeepFace: Closing the Gap to Human-Level Performance in Face Verification. In *Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on. IEEE.* 1701–1708. https://doi.org/10.1109/CVPR.2014.220

[36] Y. Umuroglu, N. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers. 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).* ACM, New York, NY, USA, 65–74. https://doi.org/10.1145/3020078.3021744

[37] S. I. Venieris and C. S. Bouganis. 2017. Latency-driven design for FPGA-based convolutional neural networks. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL).* 1–8. https://doi.org/10.23919/FPL.2017.8056828

[38] Ren W., Shengen Y., Yi S., Qingqing D., and Gang S. 2015. Deep Image: Scaling up Image Recognition. *Computing Research Repository* abs/1501.02876 (2015).

[39] J. Weston. 2016. Dialog-based Language Learning. *ArXiv:1604.06045* (April 2016). arXiv:cs.CL/1604.06045

[40] J. Weston, S. Chopra, and A. Bordes. 2014. Memory Networks. *ArXiv:1410.3916* (Oct. 2014). arXiv:cs.AI/1410.3916

[41] Xilinx. 2017. *Xilinx Zynq-7000 All Programmable SoC.* https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html.

[42] Xilinx. 2017. *Zynq UltraScale+ All Programmable Heterogeneous MPSoC.* https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html.

[43] M. Zastrow. 2016. Machine outsmarts man in battle of the decade. *New Scientist* 229, 3065 (2016), 21 –. https://doi.org/10.1016/S0262-4079(16)30458-4

[44] C. Zhang, Z. Fang, P. Zhou, P. Pan, and J. Cong. 2016. Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD).* 1–8. https://doi.org/10.1145/2966986.2967011

[45] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '15).* ACM, New York, NY, USA, 161–170. https://doi.org/10.1145/2684746.2689060

Received December 2017