2018 21st Euromicro Conference on Digital System Design

CoNNA – Compressed CNN Hardware Accelerator

Rastislav Struharik*, Bogdan Vukobratović†, Andrea Erdeljan*, Damjan Rakanović*
*University of Novi Sad, Faculty of Technical Sciences, Trg Dositeja Obradovića 6, Novi Sad, Serbia,
rasti@uns.ac.rs, andrea.erdeljan@uns.ac.rs, rdamjan@uns.ac.rs
†Kortiq GmbH, Gebrüder-Eicher-Ring 45, Forstern, Germany,
bogdan.vukobratovic@kortiq.com

***Abstract* — In this paper we propose a novel Convolutional Neural Network hardware accelerator, called CoNNA, capable of accelerating pruned, quantized, CNNs. In contrast to most existing solutions, CoNNA offers a complete solution to the full, compressed CNN acceleration, being able to accelerate all layer types commonly found in contemporary CNNs. CoNNA is designed as a coarse-grained reconfigurable architecture, which uses rapid, dynamic reconfiguration during CNN layer processing. Furthermore, by being able to directly process compressed feature and kernel maps, CoNNA is able to achieve higher CNN processing efficiency than some of the previously proposed solutions. Results of the experiments indicate that CoNNA architecture is up to 14.10 times faster than previously proposed MIT's Eyeriss CNN accelerator, up to 6.05 times faster than NullHop CNN accelerator, and up to 4.91 times faster than NVIDIA's Deep Learning Accelerator (NVDLA), while using identical number of computing units and operating at the same clock frequency.**

Keywords—machine learning, convolutional neural network, compressed CNN, hardware acceleration, FPGA

I. INTRODUCTION

Deep learning [1], and particularly Deep Neural Networks (DNNs), are currently one of the most intensively and widely used machine learning predictive models. DNNs are not new concept [2], but after recent breakthrough applications of DNNs in the fields of speech recognition [3] and image processing [4], they have returned to the academic and industrial focus. Today, different types of DNNs are being employed in wide range of applications, ranging from autonomous driving [5], medical [6], to playing complex games [7]. In many of these application domains DNNs are now able to exceed human levels of accuracy. Exceptional performance of DNNs, and in particular Convolutional Neural Networks (CNNs), predominantly arises from their ability to automatically extract high level features from raw sensory data during training phase, using large amount of data, in order to obtain effective representation of an input space.

However, superior accuracy of CNNs comes at the high cost because of their computational complexity. State-of-the-art CNNs are described by hundreds of millions of parameters and require billions of computations in order to classify single input instance [4]. It is plausible that future CNNs will be even larger, deeper, will process even larger input instances, requiring even more computations per input instance, and will be used to perform more intricate classification tasks at faster speeds, ever increasingly in real-time, within low-power operating conditions. While general-purpose compute engines, especially GPUs, have been the mainstay for much of contemporary CNN processing, increasingly there is an interest in providing more specialized acceleration platforms for the CNN computations.

Field of custom CNN hardware accelerators has been in the focus of academic community in recent years, generating more than forty different solutions. However, majority of these solutions are concerned with the acceleration of only one particular layer type from the CNN, typically the convolutional layer, because of its high computational demand, [8-12]. Several architectures have been proposed for acceleration of complete CNNs [13-15]. Furthermore, most of proposed solutions are able to accelerate only uncompressed CNNs [8-15], with only several examples being able to process compressed CNNs to some degree [16-18]. In addition, almost all proposed solutions are not highly configurable, severely limiting the flexibility of supported CNNs, in terms of supported layer types, kernel receptive field sizes, horizontal and vertical kernel stride values, pooling types, types of non-linear activation functions, etc. This is one of the main drawbacks of majority of previously proposed custom hardware solutions, which puts them at a great disadvantage when compared with CPU/GPU implementations.

In this paper we present a novel, coarse-grained reconfigurable, compressed CNN hardware accelerator, named CoNNA, which aims to overcome these difficulties. CoNNA architecture is highly configurable, enabling various configurations of compressed CNNs to be implemented efficiently. It is also capable of accelerating complete compressed CNNs, supporting all major layers found in the contemporary CNNs, like convolutional, pooling, adding and fully-connected.

The rest of the paper is organized as follows. Section II presents a brief introduction to CNNs, with emphasis on CNN compression, since CoNNA takes great benefit from CNN compression. Section III presents details of proposed CoNNA architecture, describing all of its major components. Section IV contains results of the experiments aimed at comparing performance of the CoNNA architecture with some of the previously proposed CNN hardware acceleration solutions. Finally, Section V holds final remarks and conclusions.

978-1-5386-7377-5/18/$31.00 ©2018 IEEE
DOI 10.1109/DSD.2018.00070

II. Compressed Convolutional Neural Networks

Convolutional Neural Network [19] is a type of feed-forward artificial neural network in which the connectivity pattern between the neurons is inspired by the neural connectivity found in the animal visual cortex. Individual neurons from visual cortex respond to stimuli only from a restricted region of space, known as receptive field. Receptive fields of neighboring neurons partially overlap, spanning the entire visual field. Previously it was shown that the response of individual neuron to the stimuli within its receptive field can be approximated mathematically by a 3D convolution operation, which is extensively used in CNNs.

CNN architecture is formed by stacking together layers of differentiable functions, which transform input instance into appropriate output response (e.g. holding the class scores). A number of different layers are commonly used when building a CNN: convolutional layer, pooling layer, non-linear layer, adding layer, concatenation layer, and fully-connected layer.

CNNs are both computationally and memory demanding, which makes their usage difficult, especially in embedded applications. For example, VGG-16 CNN [20] has more than 138 million network parameters. Using 16-bit number representation, approximately 276 MB of memory space must be available to store all required network parameters. Furthermore, during layer processing, CNN uses an input feature map (IFM), which is either the input image itself, or the output of previous CNN layer, and produces an output feature map (OFM). Sizes of input and output feature maps depend on the characteristics of CNN layers. For example, the largest input and output feature maps in case of the VGG-16 CNN are more than 6 MB large each. Compared to the size of the memory required to store VGG-16 CNN parameters, size of required memory for storing intermediate feature maps is only 2% of the required network parameters memory size. However, since VGG-16 CNN is composed from a total of 21 layers, required feature map data movement size to process one input image reaches 60 MB. If we analyze the number of required computations in order to classify one input image, in the case of VGG-16 CNN it reaches over 15G of MAC operations. This makes deployment of CNNs in embedded applications very challenging, especially when latency, throughput and/or power consumption is of interest.

One way of solving these issues is to reduce the memory size required to store network parameters and intermediate feature maps, by using different CNN compression methods. Generally, all CNN compression methods can be divided into two groups, depending on the target data: methods that compress CNN network parameters, and methods that compress feature map data.

One of the widely used CNN network parameter compression algorithms is the "Deep Compression" algorithm, proposed by Song Han, et al in [21]. "Deep Compression" algorithm uses a three stage pipeline to reduce the storage size required by CNN, in a manner that preserves original accuracy. First, original CNN is pruned by removing all redundant connections, keeping only the most informative ones. Next, remaining connection weights are quantized so that multiple connections share the same weight, thus only the codebook (effective weights) and the indices need to be stored. Finally, Huffman encoding is applied to take advantage of biased distribution of effective weights. After performing all three steps of the "Deep Compression" algorithm, size of memory required for storing all CNN parameters can be reduced significantly. For example, in case of VGG-16 CNN, required memory size is reduced from 276 MB to only 5.5 MB.

Please notice that pruning and weight quantization affect overall accuracy of CNN network, but as shown in [21], significant pruning and quantization can be applied to standard CNNs, trained on standard datasets, without degrading initial, unpruned, CNN network accuracy.

To further improve energy and compute efficiency, statistics of data being processed by CNN can be explored to reduce DRAM accesses using feature map compression, which is the most energy consuming operation of every CNN accelerator. Efficient feature map compression is possible because of the heavy use of ReLU activation function within CNN layers. The ReLU activation function introduces many zeros in the feature maps by rectifying all negative feature map values to zero. While number of zeros in feature maps depends on the structure of input data being fed to CNN, nevertheless it tends to increase as we move deeper into CNN. For example, in VGG-16 CNN almost 48% of the IFM values of the CONV1_2 layer are zeros on average, and this value goes up to 88% for CONV5_3 layer.

Zero Run-Length (ZRL) encoding, [12], has been proposed to efficiently exploit this phenomenon and compress zero values found in the feature maps. Consecutive runs of zeros are represented by the single run-length value, instead of original sequence of zeros. Using ZRL encoding only adds 5%-10% overhead to the theoretical entropy limit. For example, in case of VGG-16 CNN, using ZRL encoding results in reduction of required feature map transfer size per input image, from 60 MB in the uncompressed case, to only 30 MB in compressed case.

In the typical usage of feature map compression, all feature maps, except input image, are stored in compressed format in DRAM. Accelerator reads encoded IFMs from DRAM, decompresses them using appropriate decoder, and uses decompressed data stream within the accelerator to compute output feature map. Computed output feature maps are processed by the ReLU module optionally, compressed by the encoder, and transmitted to DRAM. This saves both space and R/W bandwidth of the DRAM. For example, this approach is used in MIT's Eyeriss accelerator [12]. However, a further improvement in the CNN acceleration efficiency is possible, if all unnecessary computations, originating from kernel and/or IFM zero values, are skipped. Proposed CoNNA architecture is designed specifically to exploit this opportunity.

III. CoNNA Architecture Details

Since there is a significant number of zero-valued points in CNN feature maps, because of the extensive use of ReLU activation function, and there is also a signification number of zero-valued coefficients within the kernel maps (KMs), if some variant of CNN pruning algorithm is used, an opportunity arises to optimize the evaluation process of convolutional and fully-connected CNN layers. Since these layers make heavy

use of multiplication operations between the input feature and kernel maps, a more efficient way of calculating sum-of-products, that would skip all zero-valued product terms, can be devised. Most of previously proposed CNN accelerators ignored this opportunity for optimization, and so far only a handful of proposed solutions made a partial use of it [16-18].

CoNNA architecture is specifically designed to exploit this optimization opportunity. It is able to detect on-the-fly all product terms that will result in zero-valued outcome and skip their calculation, therefore reducing the layer processing time. This is done when convolutional, pooling and fully-connected layers are being processed by the CoNNA. Furthermore, CoNNA is able to process feature and kernel maps data in the compressed form, removing the need for decompression step, thus further shortening processing time and reducing the size of on-chip memories used for storing IFM and KM data.

CoNNA architecture is designed to accelerate compressed pruned CNNs, created using different CNN pruning algorithms, similar or equal to the one from Step 1 in the "Deep Compression" algorithm [21]. As the result of the application of pruning procedure, CNN kernel maps will be sparse, since a certain number of weights will be removed from the CNN during pruning step. In contrast to "Deep Compression" algorithm, which uses weight quantization and Huffman encoding to further compress sparse kernel maps, CoNNA architecture compresses kernel maps, as well as feature maps, using a specific encoding scheme [22], which effectively removes zero values from kernel and feature maps.

CoNNA architecture is based on the idea of sequential calculation of 3D convolutions, using single Processing Block (PB) for each 3D convolution that needs to be calculated, and employing a number of PB units to calculate different 3D convolutions, one for each kernel defined for the current convolutional layer, for each IFM segment in parallel. CoNNA architecture is designed to process input feature maps and kernel maps that are compressed, using a variant of run-length compression scheme. By being able to process compressed data, significant improvement in the processing speed, and reduction in the required on-chip memory size is achieved.

Sequential approach of calculating 3D convolutions employed by the CoNNA architecture is highly flexible regarding the parameters of 3D convolutions (kernel size, horizontal and vertical stride, kernel shape, etc.), resulting in highly configurable accelerator, almost to the level of the accelerators based on CPUs and GPUs, while still having high utilization of PB units, equal or higher than the MAC utilization of the accelerators based on the custom hardware solutions. This high flexibility is in sharp contrast with most of previously proposed CNN accelerators, which are either limited to certain kernel configurations, allowing for example only 3x3 or 5x5 kernels with stride of 1. Even if they support arbitrary kernel size and stride values, their efficiency drops significantly when these parameters are set to some non-standard values.

Top level architecture of CoNNA compressed CNN accelerator is shown in the Fig. 1. CoNNA architecture is composed of following modules:

- Reconfigurable Computing Unit (RCU) – used to perform all computations defined by different layers from the CNN (including 3D convolutional, pooling, concatenation, adding layer and fully connected layer), exploiting the sparsity of kernel and feature maps.
- Input Stream Manager (ISM) – used to supply all input data (configuration, kernel map, feature map), in compressed format, coming from external DRAM memory, or on-chip cache memory, to the appropriate internal modules of the RCU module.
- Output Stream Manager (OSM) – used to format, compress and stream output feature map data to the external DRAM memory, or on-chip cache memory.

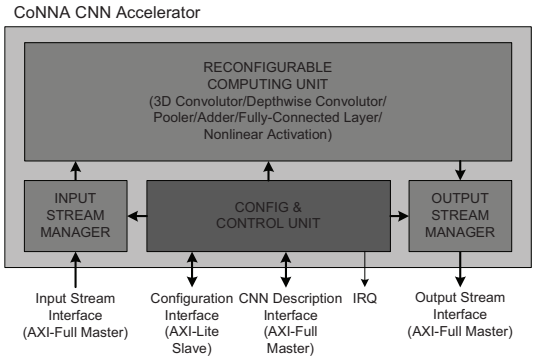

Fig. 1. CoNNA Compressed CNN Accelerator Top Level Architecture

CoNNA architecture uses four AXI interfaces to communicate with surrounding systems:

- Input Stream Interface – AXI-Full interface used to stream input data, including input image, intermediate feature maps and kernel maps for selected CNN, to the RCU module for processing.
- Output Stream Interface – AXI-Full interface used to stream output feature maps data to on-chip cache or external DRAM memory.
- CNN Description Interface – AXI-Full interface used to load structural information about the CNN that is being accelerated. CNN structural information data is organized in a linked list form, where each node defines properties of one layer from selected CNN.
- Configuration Interface – AXI-Lite interface used to configure and control CoNNA accelerator.

Since CoNNA architecture uses AXI interfaces, integration within ARM-based SoCs is greatly simplified. All data that is being processed or generated is stored in the internal on-chip cache memory or external DRAM memory, depending on available memory resources in the system. CoNNA processes CNN in sequential manner, one layer per time. It uses information about the current CNN layer to be accelerated, stored in appropriate node of associated CNN linked list, to reconfigure RCU unit and perform necessary computations. This operation is repeated until a final node in the CNN linked

list is processed, indicating the end of processing of current input instance.

Because CNN structural information is stored in a form of linked list, on-the-fly selection of target CNN to be accelerated is possible, without the need to modify and re-implement the accelerator. Furthermore, on-the-fly modification of accelerated CNN topology is possible, by removing or adding CNN layers or changing parameters of existing layers, by simple modification of target CNN linked list.

RCU module is the central module of CoNNA architecture. It is used to perform all necessary numerical calculations, defined by different CNN layers. RCU module, shown in Fig. 2, is designed as a coarse-grained reconfigurable hardware module, allowing easy, fast, dynamic, on-the-fly reconfiguration, in order to create different dataflows, optimized for processing particular CNN layer type.

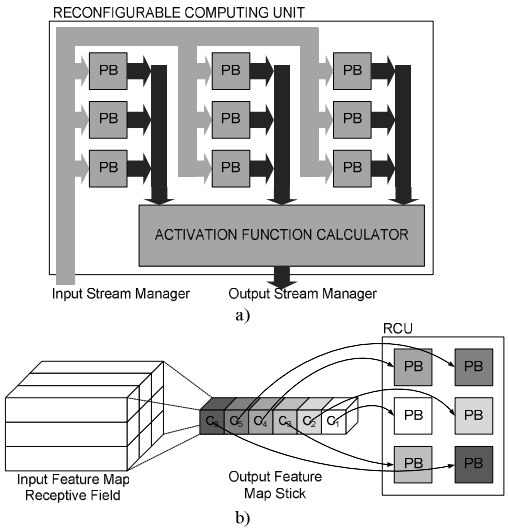

Fig. 2. a) Architecture of the RCU Module
b) Mapping 3D Convolutions on PBs

RCU module is composed from a number of Processing Blocks and one Activation Function Calculator (AFC) module. PBs are able to perform all required CNN layer processing operations, using compressed input feature map and kernel map data, including all zero product term skipping, to increase performance. AFC module is used for post-processing data coming from PB modules. AFC module contains a number of Rectified Linear Units, implementing Rectified Linear activation function and one Arbitrary Non-Linear Function Calculator unit, capable of implementing arbitrary non-linear activation functions, used in fully-connected layers.

As already mentioned, during convolutional layer processing, calculation of one complete 3D convolution is allocated to a single PB. Multiple 3D convolutions, operating on the same IFM segment are being calculated in parallel, by different PBs, as shown in Fig. 2b. In Fig. 2b distribution of 3D convolutions over available PBs, in case of processing convolutional layer with 6 different kernels, is presented. Please notice that if the number of 3D kernels is smaller than the number of PBs, some of the PBs will be inactive. Also, the amount of parallelism is limited by the number of 3D kernels. However, for most CNNs the number of 3D kernels in convolutional layers rises sharply as we move deeper inside CNN, and quickly reaches values above 100, this constraint is usually not so severe.

Detailed architecture of Processing Block is shown in Fig. 3. PB contains non-zero product term detection logic, implemented inside Data Fetcher (DF) module, which allows execution of numerical computations only when their result is different from zero, skipping all unnecessary (zero outcome) computations, resulting in significant speedup of the CNN calculation process. PB also contains four local memories, located in the Local Memory module (LM), used for storing data from selected kernel and IFM segment, in compressed format. Two memories are used to store all non-zero valued convolution kernel coefficients and their actual positions in the uncompressed kernel. Remaining two memories do the same for IFM segment that is currently being processed by the PB. Computing Unit (CU) actually performs selected compute operations on selected kernel and IFM value pairs, supplied by DF module. It contains one MAC unit, with some additional logic, for enabling different compute dataflows, depending on the type of CNN layer that is being processed by CoNNA. Output FIFO is used to synchronize processing steps performed by different PBs. Finally, Configuration Register is used to specify desired PB configuration that should be used. PB module supports several datapath configurations, which enable efficient processing of convolutional, pooling, fully-connected and adding CNN layers.

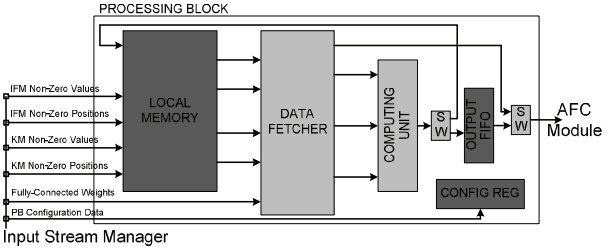

Fig. 3. Architecture of Processign Block Module

Principle of zero-valued product terms skipping employed by the PB during 3D convolution calculation is shown in the Fig. 4.

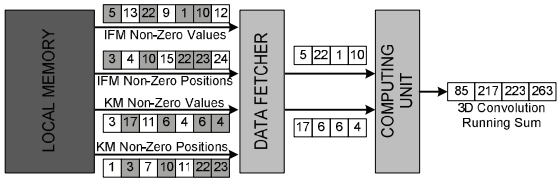

Fig. 4. Zero-Valued Product Term Skipping Principle of Operation

LM module streams information about the positions of kernel and IFM segment non-zero values to the DF module. DF module detects on-the-fly the next kernel/IFM non-zero valued pair and passes it to the CU module for processing. This is accomplished by parallel search for next coincident non-zero position in both KM and FM data streams, within a specified search window, which is 16 elements wide. This operation is

repeated until all data is used, and all relevant, non-zero valued product terms are accumulated within the CU module, to obtain the final value of 3D convolution calculation operation.

ISM module is used to stream input data to the RCU. Input stream data consists of: PB configuration data, input image data, input feature map data, and 3D kernel coefficient values or fully-connected weight values, depending on the type of CNN layer that is currently being processed. All relevant data is received from on-chip or external DRAM memory in compressed format, which is then either stored inside the Input Stick Buffer (ISB) module, in case of the input feature map data, or routed directly to the appropriate LM module inside the PB module, in case of convolutional coefficients/fully-connected weights data. Please notice that CoNNA architecture is designed to process all incoming data in compressed format, so there is no need to decompress it.

During input feature map processing by convolutional layers, majority of IFM points are being repeatedly used by adjacent 3D convolutions, as 3D convolutional kernels slide over input feature map. In order to minimize data movement between the CoNNA CNN accelerator and external DRAM memory, ISB module is used. ISB module acts as a local cache, storing selected compressed sticks, 1x1xD sections of the IFM, where D is the depth of IFM, which will be used in the upcoming 3D convolution calculation operations. Each time the same stick is needed in the 3D convolution calculation operation, instead of re-fetching it from external DRAM memory, it is fetched from the ISB module, reducing the number of data transfers from external DRAM memory, thus saving power. Once all 3D convolution operations involving a certain IFM stick are performed, that stick is removed from the ISB module and the next stick is loaded in its place from external DRAM memory. Operating principle of ISB module is illustrated on Fig. 5.

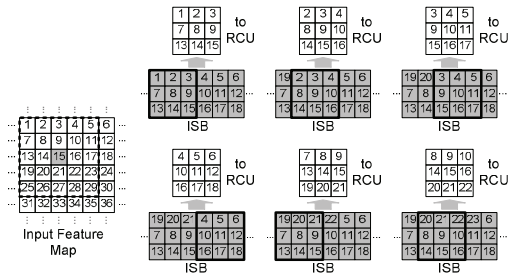

Fig. 5. Principle of ISB Operation

Fig. 5 shows a snapshot of the input feature map, 6x6 sticks in size. Please notice that there is a third dimension to the IFM, since its depth is always bigger than one, but for clarity it was omitted. If we assume that this IFM is being processed by a 3x3 kernel with horizontal and vertical strides of 1, then each IFM stick will be used in the process of calculating 9 different convolutions, as shown in the Fig. 5 for stick number 15. Without ISB module, each stick would have to be loaded from DRAM memory 9 times, but by using the ISB it is only necessary to load each stick exactly once from external DRAM. This means that in the case from Fig. 5 it is possible to reduce DRAM memory traffic, when transferring IFM data, nine times. Fig. 5 also shows the concept of replacing already used IFM sticks within the ISB module with the new ones.

OSM module is used to collect output data from the RCU module, packing it into appropriate blocks, in the form of output feature map (OFM) sticks, in order to maximize the data throughput between the accelerator and the external DRAM memory, and compressing these OFM sticks. Encoder module is used to compress sequences of zeros within the OFM sticks, which contain a larger number of zero values if the ReLU activation function is being used.

IV. Experiments

Performance of CoNNA architecture was compared to the MIT's Eyeriss CNN hardware accelerator [12], NullHop [18], and NVIDIA's open source NVDLA CNN accelerator [23], using five well-known CNNs, AlexNet [4], VGG-16, VGG-19 [17], GoogleNet [24], and ResNet-50 [25]. Eyeriss CNN hardware accelerator was chosen because it is de-facto a standard reference for comparison, used in many CNN hardware accelerator papers, NullHop accelerator is skipping zeros in input feature maps and NVLDA is one of the first commercially available accelerators coming from the industrial sector, developed by the NVIDIA, the current leader in machine learning acceleration field. Relevant data for each CNN used in experiments is presented in Table I.

TABLE I. Characteristics of CNNs used in Experiments

CNN Architecture	# of Layers (C,P,A,FC)	# Parameters [Millions]	# Operations [GOp]
AlexNet [4]	(16,5,0,3)	60.93	1.45
VGG-16 [20]	(16,5,0,3)	138.35	30.95
VGG-19 [20]	(19,5,0,3)	143.66	39.28
GoogleNet [24]	(21,5,0,1)	6.97	3.16
ResNet50 [25]	(49,2,16,1)	25.50	7.72

Configurations of Eyeriss, NullHop and NVDLA CNN hardware accelerators that were used in performance comparison experiments are presented in the Table II. Configurations of Eyeriss and NullHop accelerators are the ones that were used in [12], [18], in order to enable easy comparison. As for the NVDLA [23], it comes with an Excel sheet estimator, which allows specifying different configuration options. We have used the one presented in Table II, since CoNNA accelerator is dominantly intended for usage in low-cost, edge devices.

TABLE II. Important Characteristics of Reference CNN Accelerators used for Comparisson

Parameter/Accelerator	Eyeriss [12]	NullHop [18]	NVLDA [23]
Number of MAC Units	168	128	32
Operating Frequency	200 MHz	60 MHz	100 MHz
Arithmetic Precission	16-bit	16-bit	16-bit

In order to perform comparison experiments, CoNNA compressed CNN accelerator has been configured in three different configurations, specified in Table II. The idea was to use identical configurations as the reference accelerators, in terms of number of computing elements (MAC units), operating frequency and number representation, to enable as fair as possible performance comparison. Please notice that this was possible, because the CoNNA architecture was modeled as

a soft-IP core, using SystemVerilog HDL, with many configuration parameters, allowing easy generation of different instances of the same underlying architecture.

All three instances of the CoNNA architecture have been implemented targeting Xilinx ZU9 MPSoC device. Vivado Design Suite 2017.3 has been used to perform synthesis and implementation, with default settings. Implementation results for three different configurations of CoNNA accelerator that have been used in the experiments are shown in the Table III.

TABLE III. IMPLEMENTATION RESULTS FOR THREE CONNA CONFIGURATIONS USED IN EXPERIMENTS

Parameter/ Configuration	CoNNA_C1	CoNNA_C2	CoNNA_C3
Slice LUTs	142470	112550	35042
Number of BRAMs	364	284	86
Number of DSPs	177	137	41
Operating Frequency	200 MHz	60 MHz	100 MHz

From Table III it can be seen that all configurations have been successfully implemented on Xilinx ZU9 MPSoC device. The largest one, CoNNA_C1, uses 52% of available LUTs, 40% of available BRAMs and 7% of available DSP blocks, and is actually able to operate at the peak clock frequency of 280 MHz, well above 200 MHz required for comparison with the Eyeriss architecture. Please notice that the number of used DSP blocks for all three configurations is actually larger than 168, 128 and 32 MACs used by Eyeriss, NullHop and NVDLA respectively, since CoNNA uses additional DSPs inside ISM module, but these additional DSP blocks are not used to perform computations defined by the target CNN.

All three implementations of CoNNA architecture have been tested and benchmarked on five selected CNN architectures (AlexNet, VGG16, VGG19, GoogleNet and ResNet-50) using Xilinx ZCU102 development board. Based on these experiments performance results have been extracted and compared to three selected reference CNN accelerator architectures (Eyeriss, NullHop, NVDLA).

Results of performance experiments are presented in the Table IV. For each of the reference accelerators, performance data in terms of frame latency, frame rate, average compute power and efficiency is presented. Each of the reference accelerators is followed by appropriate configuration of the CoNNA architecture, which was configured in the same way as the reference accelerator to allow for fair comparison. Performance data for Eyeriss and NullHop accelerators was taken from papers that introduced these architectures [12], [18]. For NVDLA accelerator supplied Excel sheet estimator has been used to generate achievable performance data.

In case of Eyeriss accelerator, performance data is related to processing convolutional layers only, since Eyeriss cannot accelerate pooling and fully-connected layers. Also, frame latency is reported only when Eyeriss processes input images in batches, with batch sizes of 4 and 3, for AlexNet and VGG16 CNNs respectively. Performance data for NullHop and NVDLA is related to processing complete CNNs, since both of these architectures, as well as CoNNA, can process all layers from CNN. Also, reported frame latencies in these cases relate to the latency of processing a single input image.

Compute performance for all architectures has been calculated as the total number of operations needed to process one input image by the target CNN, multiplied by the time required to process one input image. Effective efficiency was calculated as the achievable compute performance divided by the theoretical peak compute performance (calculated as the number of available MAC units multiplied by the operating frequency).

TABLE IV. PERFORMANCE COMPARISON WITH REFERENCE CNN ACCELERATORS

Architecture	CNN Architecture	Latency [ms/frame]	Frame Rate [frames/s]	Performance [GOp/s]	Effective Efficiency [%]
Eyeriss[a]	AlexNet	115.3	34.69	46.14	68.66%
	VGG16	4309.5	0.69	21.36	31.79%
CoNNA_C1[a]	AlexNet	14.20	70.40	93.74	139.50%
	VGG16	102.77	9.73	301.17	448.17%
NullHop	VGG16	2269.00	0.44	13.65	88.87%
	VGG19	2439.00	0.41	16.10	104.82%
CoNNA_C2	VGG16	375.94	2.66	82.33	536.03%
	VGG19	436.68	2.29	89.94	585.56%
NVDLA	AlexNet	263.01	3.80	5.51	86.09%
	GoogleNet	384.76	2.60	8.21	128.28%
	ResNet50	1843.51	0.54	4.17	65.16%
CoNNA_C3	AlexNet	96.90	10.32	14.97	233.91%
	GoogleNet	202.02	4.95	15.63	244.18%
	ResNet50	431.03	2.32	17.91	279.84%

(a) Accelerating only convolutional layers

From Table IV it can be seen that CoNNA architecture outperforms all reference CNN accelerator architectures. When compared with the MIT's Eyeriss CNN accelerator, from Table IV it can be seen that the CoNNA architecture achieves higher frame rates and higher efficiency for both CNN networks used. In case of AlexNet CNN acceleration, CoNNA is capable of reaching 2.03 times higher frame rate than the Eyeriss. In case of VGG16 CNN acceleration, this improvement is even higher, CoNNA is able to reach 14.10 times higher frame rate than the Eyeriss. Also it can be seen that CoNNA is able to achieve much higher effective efficiency, even above 100% due to the compressed data processing, than the Eyeriss. The main reason for this improvement lies in the fact that the CoNNA architecture is able to take advantage of compressed kernel/weight and feature maps processing and Eyeriss is not.

When compared to NullHop, CoNNA is also able to achieve better performance, but this time improvement is not as dramatic as in the case of Eyeriss. CoNNA achieves 6.05 and 5.58 times faster frame rates than the NullHop, when processing VGG16 and VGG19 CNNs respectively. The main reason for this is that the NullHop accelerator is being able to take advantage of the sparsity present in the input feature maps. But because NullHop is not able to take advantage of the sparsity of CNN weights, because it is not designed to process compressed CNNs, its performance is lower than CoNNA's. NullHop is also able to reach effective efficiency values greater than 100%, due to the fact that it is able to skip all computations where input feature map values are equal to zero.

Finally, from Table IV it can be seen that CoNNA architecture is also superior to the NVDLA, for all three CNNs used in experiments. CoNNA is able to reach 2.72, 2.15 and 4.91 times higher frame rates than NVLDA, when accelerating AlexNet, GoogleNet and ResNet-50 CNNs respectively. However, the improvement over NVLDA is smaller than in the case of Eyeriss and NullHop, because NVLDA is using Winograd algorithm to speed-up the 3D convolution calculation process.

In order to better understand how CoNNA processes CNNs, and where the bottlenecks are, Table V and Fig. 6 present detailed execution information for CoNNA_C3 configuration, when it is accelerating AlexNet CNN.

TABLE V. CONNA_C3 PERFORMANCE BREAKDOWN FOR ALEXNET CNN

Layer	Active PBs	Total Latency [us]	Processing Latency[us]	PB Efficiency [%]	Number of Operations [GOp]	Performance [GOp/s]
CONV1	32 (100%)	60689.63	27668.73	45.60%	0.21083	3.47
POOL1	4 (12.5%)	2274.32	1574.48	8.65%	0.00126	0.55
CONV2_1	32 (100%)	9049.99	8151.04	98.12%	0.22395	24.75
CONV2_2	32 (100%)	9049.99	8151.04	98.12%	0.22395	24.75
POOL2	4 (12.5%)	1029.36	596.72	7.24%	0.00078	0.76
CONV3	32 (100%)	5409.80	4497.29	83.13%	0.29904	55.28
CONV4_1	32 (100%)	1700.80	1342.00	78.90%	0.11214	65.93
CONV4_2	32 (100%)	1700.80	1342.00	78.90%	0.11214	65.93
CONV5_1	32 (100%)	1207.34	968.14	80.19%	0.07476	61.92
CONV5_2	32 (100%)	1207.34	968.14	80.19%	0.07476	61.92
POOL3	4 (12.5%)	138.61	46.45	4.19%	0.00016	1.15
FC6	4 (12.5%)	1943.70	1902.74	12.24%	0.07550	38.84
FC7	4 (12.5%)	886.74	845.78	11.92%	0.03355	37.83
FC8	4 (12.5%)	583.58	573.58	12.29%	0.00819	14.03

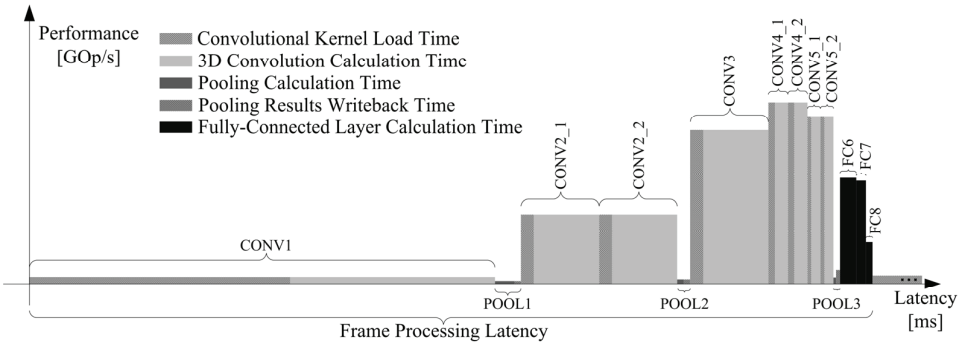

Fig. 6. Layer Latency Brakedown for AlexNet CNN, Processed by CoNNA_C3 Configuration

Table V presents the following data for each AlexNet CNN layer. Active PB field holds the number of Processing Blocks that are actually being used to process current CNN layer. Total latency stands for the total time needed to process a CNN layer, including time required to preload CNN weights and store the final results. Processing latency is the time that is actually spent performing all computations defined in the current CNN layer, excluding any preparatory steps and is therefore always shorter than the Total latency. PB efficiency value is calculated as the ratio of Processing latency and Total latency, multiplied with the percentage of active PBs for the current CNN layer. Number of operations is the total number of arithmetic operations required to be performed in order to process current CNN layer. Performance is the computational throughput for the current CNN layer, calculated as the ratio of Number of operations and Total latency.

From Table V and Fig. 6 it can be seen that CoNNA architecture is being able to maintain high utilization of PBs over all convolutional layers, irrelevant of their size and kernel characteristics. Efficiency of processing convolutional layers is extremely important, since they constitute the majority of compute operations in modern CNNs. It can also be seen that, as we move to deeper convolutional layers of the CNN, PB efficiency starts to drop. This is because more time has to be spent in preloading kernel coefficient values into PBs, before actual computation can start. The reason for this is that the depths of convolutional layers increase as we go deeper in the CNN, resulting in more kernel weights data per each 3D convolution that has to be preloaded into PBs. Since CoNNA

uses a limited bandwidth data bus to transfer data to the PBs, it takes increasingly more time to preload kernel data as the size of 3D kernels increases.

From Table V and Fig. 6 it can also be seen that PB efficiency drops significantly when CoNNA is processing pooling and fully-connected layers. Processing of these layer types is data movement intensive, so available data bandwidth of CoNNA's data bus is the main limiting factor. However, these layers are not computationally intensive, so this inefficiency is not significantly degrading the total performance of CoNNA architecture. From Table V it can be seen that the number of operations for all pooling and fully-connected layers in the case of AlexNet CNN constitutes only 8% of the total number of computations.

V. Conclusion

In this paper a novel CNN hardware accelerator has been proposed. CoNNA is coarse-grained reconfigurable hardware architecture capable of accelerating complete CNNs. It can be used to accelerate convolutional, pooling, fully-connected, concatenation and adding layers of the target CNN. CoNNA is designed to process compressed CNNs, as well as input feature maps, which seems to enable achieving higher performance values, when compared to some of the previously proposed CNN accelerator solutions. CoNNA architecture, when configured to use identical number of MAC units and run at the same operating frequency as previously proposed MIT's Eyeriss, NullHop and NVIDIA's NVDLA CNN accelerators, enables up to 14.10, 6.05 and 4.91 times faster CNN execution of standard CNN architectures respectively.

References

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning", Nature, vol. 521, no. 7553, pp. 436–444, May 2015.

[2] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," Biological Cybernetics, vol. 36, no. 4, pp. 193–202, 1980.

[3] L. Deng, J. Li, J.-T. Huang, K. Yao, D. Yu, F. Seide, M. Seltzer, G. Zweig, X. He, J. Williams, and Y. Gong, "Recent advances in deep learning for speech research at microsoft", IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 8604–8608, 2013.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks", Advances in Neural Information Processing Systems, pp. 1097-1105, 2012.

[5] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "Deepdriving: Learning affordance for direct perception in autonomous driving," in Proceedings of the IEEE International Conference on Computer Vision, pp. 2722–2730, 2015.

[6] A. Esteva, B. Kuprel, R. A. Novoa, J. Ko, S. M. Swetter, H. M. Blau, and S. Thrun, "Dermatologist-level classification of skin cancer with deep neural networks," Nature, vol. 542, no. 7639, pp. 115–118, 2017.

[7] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search", Nature, vol. 529, no. 7587, pp. 484–489, Jan. 2016.

[8] C. Zhang, L. Peng, S. Guangyu, G. Yijin, X. Bingjun, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks", Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 161-170, February 2015.

[9] K. Guo, S. Lingzhi, J. Qiu, S. Yao, S. Han, Y. Wang, and H. Yang, "Angel-eye: A complete design flow for mapping cnn onto customized hardware", 2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), pp. 24-29, July 2016.

[10] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs", Proceedings of the 54th Annual Design Automation Conference, p. 29, June 2017.

[11] C. Wang, L. Gong, Q. Yu, X. Li, Y. Xie, and X. Zhou, "DLAU: A scalable deep learning accelerator unit on FPGA", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 36, no. 3, pp. 513-517, 2017.

[12] Y.H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks", IEEE Journal of Solid-State Circuits, vol. 52, no. 1, pp. 127-138, 2017.

[13] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J. S. Seo, and Y. Cao, "Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks", Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 16-25, February 2016.

[14] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, "Going deeper with embedded fpga platform for convolutional neural network", Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 26-35, February 2016.

[15] Z. Liu, Y. Dou, J. Jiang, J. Xu, S. Li, Y. Zhou, and Y. Xu, "Throughput-Optimized FPGA Accelerator for Deep Convolutional Neural Networks", ACM Transactions on Reconfigurable Technology and Systems, vol. 10, no. 3, p 17, 2017.

[16] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos. "Cnvlutin: ineffectual-neuron-free deep neural network computing", 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), pp. 1-13, June 2016.

[17] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: efficient inference engine on compressed deep neural network", Proceedings of the 43rd International Symposium on Computer Architecture, pp. 243-254, June 2016.

[18] A. Aimar, H. Mostafa, E. Calabrese, A. Rios-Navarro, R. Tapiador-Morales, I.A. Lungu, M.B. Milde, F. Corradi, A. Linares-Barranco, S.C. Liu, and T. Delbruck, "NullHop: A Flexible Convolutional Neural Network Accelerator Based on Sparse Representations of Feature Maps", arXiv preprint arXiv:1706.01406, 2017.

[19] Y. LeCun, K. Koray, and F. Clément, "Convolutional networks and applications in vision", Proceedings of 2010 IEEE International Symposium on Circuits and Systems (ISCAS), pp. 253-256, May 2010.

[20] K. Simonyan, and A. Zisserman, "Very deep convolutional networks for large-scale image recognition", arXiv preprint arXiv:1409.1556, 2014.

[21] H. Song, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding", arXiv preprint arXiv:1510.00149, 2015.

[22] A. Erdeljan, B. Vukobratović, R. Struharik, "IP Core for Efficient Zero-Run Length Compression of CNN Feature Maps", 25th Telecommunications Forum TELFOR 2017, Belgrade, Serbia, November 21-22 2017.

[23] NVIDIA Deep Learning Accelerator, [Online]. Available: http://nvdla.org/

[24] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions", Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 1-9, June 2015.

[25] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition", Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 770-778, June 2016