Microprocessors and Microsystems 60 (2018) 151–161

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

Throughput optimizations for FPGA-based deep neural network inference

Thorbjörn Posewsky[a], Daniel Ziener*,[b]

[a] Institute of Embedded Systems, Hamburg University of Technology (TUHH), Hamburg 21073, Germany
[b] Computer Architectures for Embedded Systems, University of Twente, Enschede 7500 AE, The Netherlands

ARTICLE INFO

Keywords:
Deep neural networks
Batch processing
Pruning
Compression
FPGA
Inference
Throughput optimizations
Fully-connected

ABSTRACT

Deep neural networks are an extremely successful and widely used technique for various pattern recognition and machine learning tasks. Due to power and resource constraints, these computationally intensive networks are difficult to implement in embedded systems. Yet, the number of applications that can benefit from the mentioned possibilities is rapidly rising. In this paper, we propose novel architectures for the inference of previously learned and arbitrary deep neural networks on FPGA-based SoCs that are able to overcome these limitations. Our key contributions include the reuse of previously transferred weight matrices across multiple input samples, which we refer to as batch processing, and the usage of compressed weight matrices, also known as pruning. An extensive evaluation of these optimizations is presented. Both techniques allow a significant mitigation of data transfers and speed-up the network inference by one order of magnitude. At the same time, we surpass the data throughput of fully-featured x86-based systems while only using a fraction of their energy consumption.

1. Introduction and motivation

For more and more people, *Deep Neural Network* (DNNs) have become a substantial part of their everyday life. Applications like image classification [1] or speech recognition [2] are used by millions on their wearables, smartphones, or tablets. This applies not only to mobile computing, it also holds true for related areas like computer vision or robotics. However, these emerging areas face restrictive power requirements and limited processing power, in contrast to high-performance computing, which is more often associated with deep learning techniques.

In order to achieve state-of-the-art and beyond classification rates in tasks like object recognition, the number of artificial neurons and layers in DNNs has grown to ever new records in the past years. Aside from a significantly increased demand for computational power, the size needed to store such networks has similarly increased. For embedded devices, this is particularly challenging since memory is typically a scarce resource and, more importantly, the access to off-chip memories represents the dominating factor when considering the energy consumption [3]. Hence, to lower both DNN inference time and energy-consumption, this work focuses on techniques that reduce the amount of data to be transferred.

The first technique, called *batch processing*, originates from applications that use or even require *multiple* inferences of DNNs with *similar inputs* (also referred as *samples*) before proceeding to the next step. For example, the movement of UAVs, robots, or autonomous cars requires that images from different directions are evaluated before the next move is determined [4]. Deploying speech recognition at scale (i.e. in data centers) is another example where a study [5] reports that a sequential processing of requests is inefficient due to the memory bound as well as a limited amount of exploitable parallelism. Instead, grouping multiple samples together and processing this so-called batch can often significantly increase throughput in cases where several DNN inferences are necessary or a small latency increase is tolerable.

The second technique investigated in this work, now known as *pruning*, represents a form of DNN compression [6] [3]. Instead of reusing data as in batch processing, pruning reduces the number of synaptic connections to other neurons such that the overall amount of data is reduced. As described before, the tendency of growing DNNs also increases the possibility of redundant connections. Here, pruning can help eliminate these connections with minor, if any, accuracy drops for tasks such as classification.

While batch processing is a standard technique for an efficient DNN training [7] (called mini-batch processing in the context of stochastic gradient descent), it is rarely used for the inference of DNNs. In [8], we showed how this concept affects the design of hardware accelerators for the inference of DNNs (*forward-propagation*) and what latency consequences are imposed by realizing this concept in dedicated hardware.

Similarly, only a very limited number of previous works exists considering hardware-based support for pruned DNNs. In this paper, we extend [8,9] and show how a complete streaming architecture for arbitrarily pruned DNNs can be designed as opposed to designs with

* Corresponding author.
E-mail address: d.m.ziener@utwente.nl (D. Ziener).

https://doi.org/10.1016/j.micpro.2018.04.004
Received 2 June 2017; Accepted 12 April 2018
Available online 25 April 2018

partially or completely embedded parameters.

The contribution of this paper includes all the above mentioned aspects using an embedded FPGA-based SoC with limited external memory bandwidth. Furthermore, we show for the first time an extensive evaluation and direct comparison of both techniques using the same DNN networks and data sets. This includes for both techniques and designs expectable

- throughput gains,
- accuracy variations, and
- hardware design consequences and restrictions of complete *streaming* architectures.

We focus particularly on an efficient inference of *fully-connected* DNNs since these layers are the most memory-intensive and build the foundation for all of today's most successful network kinds.

The rest of this paper is organized as follows: Section 2 gives an overview of different network types, optimizations and corresponding hardware designs. Section 3 provides some background information for neural network processing. The concepts and architectures of our accelerators are explained in Sections 4 and 5, respectively. Section 6 continues with experimental results for different hardware configurations, software platforms, and using several network architectures. Finally, Section 7 concludes the work and highlights future research directions.

2. Related work

In the past two decades, several hardware accelerators for various kinds of neural networks were introduced. Many, and in particular early works, target shallow network architectures with few neurons or synaptic connections. Two comprehensive studies that compare designs implemented on FPGAs or as ASICs are given in [10,11]. While these works serve the purposes of their time, today they are no longer applicable or optimized for networks of the deep learning era since the number of hardware neurons or connections is no longer sufficient.

An accelerator that addresses these deep networks is presented in [12]. It is based on an array of so called *Neural Processing Unit* (NPUs) that are used to compute the majority of involved operations (e.g., vector-matrix operations) in parallel. Although this approach uses similar to our batch processing design a *Time Division Multiplexing* (TDM) processing scheme with a fast hardware-based switch of layers, it only exploits parallelism for one sample and relies on an Ethernet connection for the transfer of network stimuli. This requires a very time consuming retransfer of the required weight matrices for every sample and is not directly deployable for mobile devices.

Recently, many accelerator designs for *Convolutional Neural Network* (CNNs) were introduced. CNNs are often found in image and video recognition systems and typically use a series of kernels or convolution matrices prior to the above mentioned fully-connected network architecture [13]. Since the number of parameters for convolution matrices is typically only a fraction of the weights of fully-connected network layers, the exploitable compute parallelism is usually greater and thus favors hardware accelerators. A typical design that addresses these networks is called *NeuFlow* and proposed in [14,15]. It relies on a two-dimensional grid of *Processing Tile* (PTs) instead of a one-dimensional array of NPUs. This resembles the concept of a systolic array, but both the routes of the dataflow and the operation of the individual PTs are reconfigurable. However, as reported in [16], the proposed design has scalability issues which is problematic for batch processing as shown in Section 6. As a consequence, a CNN design with a linear array of processing elements (called collections) is shown in [16,17], respectively. Nonetheless, both designs are particularly designed to accelerate CNNs. Internal buffer and routing elements for an efficient execution of multiple samples in fully-connected layers are missing.

A third important type of networks is known as *Recurrent Neural Network* (RNN) [13]. RNNs allow the processing of input sequences through cyclical connections in the network architecture. Like fully-connected layers, these networks are typically memory bound and thus make a parallel execution more difficult. Consequently, corresponding designs are less frequent. However, an early approach for a state-of-the-art RNN, called LSTMs, using the same FPGA as in this work is shown in [18].

The theoretical foundation for our second accelerator with support for pruned neural networks was introduced by LeCun et al. in [6]. Originally, it was used to improve generalization and speed of learning in shallow network architectures. However, Han et al. [3] recently revived the technique for DNNs and were able to reduce the number of connections by a factor between 9x and 13x. Although the pruned networks included both convolutional and fully-connected layers, most connections could be removed in the memory-intensive fully-connected layers. Furthermore, they also introduced a form of parameter quantization and a subsequent Huffman encoding for the pruned and quantized networks. A corresponding ASIC design with large on-chip memories for the remaining parameters after pruning and quantization (without Huffamn encoding) is given in [19]. As discussed later, our accelerator utilizes a similar format, presented in [20], for the resulting sparse matrices (e.g., after pruning) but does not embed parameters for specific DNNs on-chip. Instead, we propose a streaming architecture for arbitrary DNNs. Very recently their approach was further extended to support LSTMs for speech recognition on high-performance FPGAs [21]. Compared to our design, they use more complex DNNs for specific tasks and directly map them onto large FPGAs (up to 10x larger than the one used in this work). Instead, our design focuses on embedded FPGA-based SoCs with very limited on-chip memory resources and much slower memory interconnects. We specifically design interfaces between the SoC's processors, off-chip memory and FPGA in order to optimize DNNs on such low-end and low-power devices.

3. Background

A typical neural network contains several layers $j = 1 \cdots L$, where L denotes the number of layers. A layer j itself consists of s_j neurons. As already mentioned, one major goal of this work is to accelerate the processing of fully-connected layers in DNNs. These layers are characterized by a bipartite graph of neuron connections between two adjacent layers j and $j+1$ for $1 \leq j \leq L-1$. For the rest of this work, we will specify the architecture of these networks through the number of neurons s_j in each layer. For example, a network with $L = 3$ layers is denoted by $s_0 \times s_1 \times s_2$. The synaptic strength of a connection is modeled through a scalar value $w_{i,k}^{(j)}$ called *weight* that represents the connection to the ith neuron in layer $j+1$ from the kth neuron in layer j. A transition from layer j to the next layer $j+1$ involves a *weight matrix* $W^{(j)}$ where $w_{i,k}^{(j)}$ are the components. The number of rows in $W^{(j)}$ equals the number of neurons s_{j+1} in layer $j+1$ and the number of columns corresponds to the number of neurons s_j in layer j. Fig. 1 gives

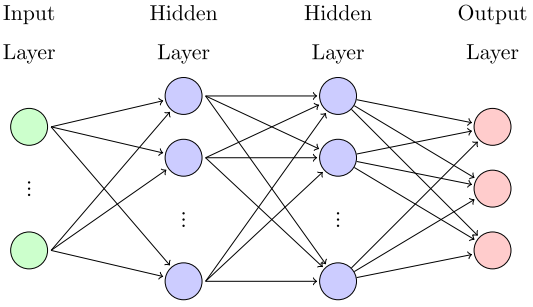

Fig. 1. Example neural network with two hidden layers ($L = 4$). In this case, the output layer $j = 4$ contains three neurons, whereas all previous layers $j = 1 \cdots 3$ contain an arbitrary number of neurons.

an example of a neural network with four layers.

The result of each neuron is computed by the following two functions: First, the *transfer function* which is a series of *multiply-accumulate* (MAC) operations of the outputs $a_k^{(j)}$ of connecting neurons in the layer j and their corresponding weights $w_{i,k}^{(j)}$:

$$z_i^{(j+1)} = \sum_{k=0}^{s_j} w_{i,k}^{(j)} \cdot a_k^{(j)}$$

Second, a subsequent application of a non-linear function, called *activation function* φ, with the result of the transfer function as argument:

$$a_i^{(j+1)} = \varphi(z_i^{(j+1)})$$

The outputs of this function are also referred to as activations for the sake of brevity. A variety of different types of activation functions φ are known in neural network literature. For example, while before the deep learning era the so called *sigmoid* function was found most frequently, today's most successful implementations usually deploy *Rectified Linear Unit* (ReLU) [22] or variations of it [23]. It is also not uncommon to utilize different functions in the same neural network, e.g., the sigmoid function for the output layer and ReLUs for all other layers. In order to ensure the application of our accelerator for various trained networks, the accelerator is able to choose between different functions at runtime.

4. Concept

On the hardware side, modern FPGAs typically offer a rich set of DSP and RAM resources within their fabric that can be used to process these networks. However, compared to the depth and layer size of deep neural networks, these resources are no longer sufficient for a full and direct mapping the way it was often done in previous generations of neural network accelerators. For example, consider a network with $L = 7$ layers and architecture $784 \times 2500 \times 2000 \times 1500 \times 1000 \times 500 \times 10$ that was proposed in [24]. The number of neurons is 8294 and the total size of the network weights is approximately 22MB if each weight is encoded using 16 bits. Compared to FPGA platforms like the Zynq, where even the largest device is limited to 2020 DSP slices and a total BRAM size of less than 3MB [25, pp. 622], a complete mapping with all neurons and weights directly onto the FPGA is no longer possible. Here, new algorithms and architectural adaptions are required in order to enable the inference of DNNs on platforms with such limited resources.

Modern and deep neural networks are usually partitioned into smaller *sections* in order to process them on embedded FPGAs platforms. We refer to a *section* as a certain number m of neurons in a given layer j with $m \leq s_{j+1}$ that can be processed in parallel through our hardware coprocessor with m individual *processing units*. Each processing unit is responsible for the transfer function of exactly one neuron in each section. By applying a time division multiplexing scheme, the whole network can be processed on these m processing units and a subsequent activation function. Each *processing unit* may consists of r different computation resources, e.g., multipliers which are able to consume r weights as inputs in parallel for the calculation of the transfer function. The number of processing units m and the corresponding number of compute resources per processing unit r indicate the degree of parallelism and depends on the number of available compute resources in hardware. Since the network is fully-connected, the computation of layer j requires that all previous layers $1 \cdots j-1$ are *completely* processed. Consequently, a hardware implementation can only use parallelism in the current layer j and not across multiple layers.

Due to the fact that the on-chip memory is not sufficient for storing all needed weights for an arbitrary layer, only the weights for processing the current section can be loaded from external memory. When comparing the size of the input data (s_j values), the output data (m values), and in particular the weights ($\approx s_j \times m$ values), it can be seen that the transfer of the weight matrix is very costly. Three concepts for reducing memory data transfers are discussed in the following: Weight encoding in order to reduce the used weight bits, batch processing to reuse weights which are already on-chip [8], and pruning to remove weights such that it is unnecessary to transfer them.

4.1. Weight encoding

An enormous impact on the throughput, complexity, and amount of needed memory resources has the encoding of the weight matrices $W^{(j)}$ with the corresponding individual weights $w_{i,k}^{(j)}$. Software-based implementations often use floating point weights, whereas the most hardware implementations using fixed point representations. On hardware implementations, the encoding format can be freely chosen. Accuracy evaluations show that often the accuracy loss through weight bit reduction is negligible compared to the advantages of increased weight memory throughput and reduced operation complexity [14,16,26]. Extreme approaches reducing the weights to only a single bit which is called binary neural networks (BNNs) processing [27]. However, the accuracy of such BNNs is relatively low compared to other approaches. Reducing the used weight bits has mainly the advantage of reduced data amount for storing and transferring weights. Furthermore, the computation complexity might be reduced, e.g., transforming the multiplication into an addition on the BNN approach. However, due to the often usage of hardware multipliers or DSP resources with fixed input width, only a small reduction of used resources is achievable. Most hardware implementations are able to perform every clock cycle one operation, i.e. multiplication. Therefore, the reduction of the used weight bits do not increase the overall throughput of the processing elements, if the weight transfer is ignored.

4.2. Batch processing

A straightforward section by section processing for just one sample has the drawback of exchanging the costly transferred weights for every new section. This huge demand of memory transfers turns the interface to the external memory into the major bottleneck for fully-connected layer processing. The input data $a_k^{(j)}$ (i.e., results of the previous layer), however, is needed for all sections in the current layer. Therefore, it should be cached in on-chip memories during the complete processing. The main contribution of our batch processing approach is the reuse of already transferred and stored weights for one section by processing different input samples through time division multiplexing. This processing scheme is visualized in Fig. 2.

Given a batch of n different input samples, the algorithm starts by processing all first sections of the n samples before proceeding to all second sections of the n samples. Thereby, all iterations $1 \cdots n$ use the same set of weights, however, distinct input samples. Only before the processing of the second sections, a new set of weights is transferred. This technique reduces the amount of memory transfers significantly and, therefore, mitigates the memory interface bottleneck. Note that for general matrix operations, similar processing schemes were already discussed in earlier works [28]. However, as shown in Section 5, our design specifically incorporates all DNNs operations and allows an interleaving of this concept and subsequent operations (i.e., activation functions) in order to further enhance the throughput.

4.3. Pruning

In order to reduce the amount of data to transfer from the memory and for calculation, it is possible to remove some connections entirely. After some initial iterations of the training phase, small weights which are below a certain threshold δ can be set to zero:

Subsequently, these pruned weights are kept at zero and the remaining weights are refined in the following iterations of the training phase. While this can potentially reduce the accuracy if too many weights are pruned, it was shown that over 90% of the weights in fully-connected layers of common CNNs can be pruned without noticeable

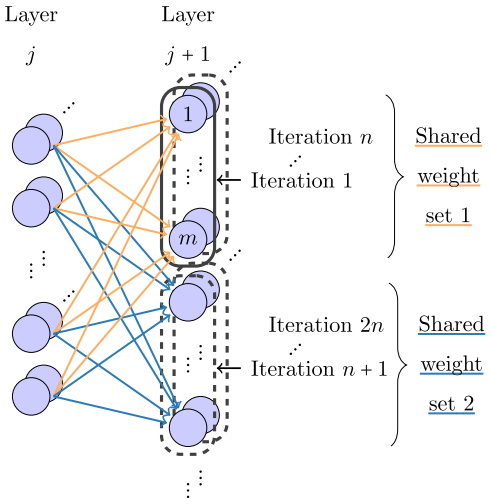

Fig. 2. Conceptual batch processing with batch size n and section size m. All m neurons in a section are processed in parallel. The first section of all n samples shares the same collection of weights. The second section of all n samples shares the next collection of weights, and so on.

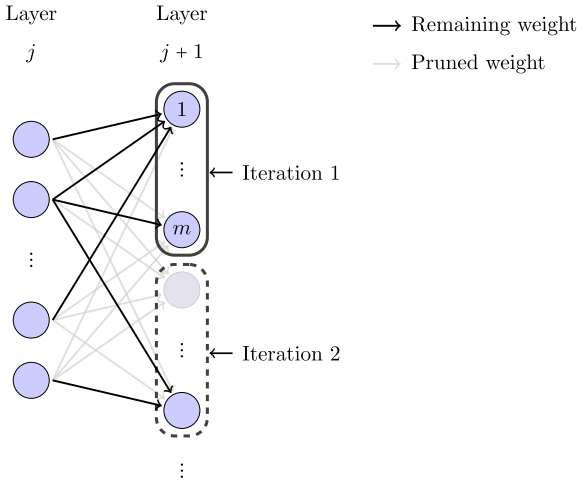

Fig. 3. Example of a pruned DNN layer. Up to m neurons in a section can be processed in parallel. Computations are only required for the remaining weights and can be entirely skipped for neurons with only pruned weights.

accuracy drops [3]. An example of a pruned layer is shown in Fig. 3.

Since weights with the value zero neither influence the result of the transfer nor the result of the activation function, these weights don't have to be stored in memory, transferred, and used for computations. However, by pruning weights, the weight matrix becomes sparse and the hardware needs to be designed in a way that the involved calculations are computed efficiently. Additionally, this presupposes a suitable format to store the sparse weight matrices in a continuous way with the smallest possible footprint. Details about sparse matrix computation and storage are further discussed in Section 5.

4.4. Throughput discussion

Assuming that the computation resources of a general hardware architecture are able to process every clock cycle one value, the following amount of clock cycles are needed for the computation of layer $j+1$ with s_{j+1} neurons and s_j input activations for a total of N input samples:

$$\left\lceil \frac{s_{j+1}}{m} \right\rceil \cdot \left\lceil \frac{s_j \cdot \left(1 - q_{\text{prune}}^{(j)}\right)}{r} \right\rceil \cdot N$$

whereas m is the number of neurons which can be processed in parallel, and r is number of parallel processed operations per neuron. The pruning factor $0 \leq q_{\text{prune}}^{(j)} \leq 1$ expresses the reduction of weights by pruning. For example, if 90% of all weights $w_{i,k}^{(j)}$ are pruned, then the pruning factor is $q_{\text{prune}}^{(j)} = 0.9$. Through elaborated pipelining, most hardware architectures achieve a throughput of one value per computation resource, just as well as our architectures, presented in Section 5. For large s_{j+1}, s_j, and N, we can calculate the approximated processing time:

$$t_{calc} \approx \frac{s_{j+1} \cdot s_j \cdot N \cdot \left(1 - q_{\text{prune}}^{(j)}\right)}{m \cdot r \cdot f_{pu}},$$

whereas f_{pu} is the clock frequency of the processing units.

However, this approximation does not consider the transfer time of the weight matrix $w^{(j)}$ from the external memory. The time to transfer all weights for the calculation of layer $j+1$ for $N \gg n$ input samples is

$$t_{\text{mem}} = \frac{s_{j+1} \cdot s_j \cdot b_{\text{weight}} \cdot N}{T_{\text{mem}} \cdot n},$$

where n is the batch size, b_{weight} is the size of each weight, and T_{mem} the actual memory throughput. If weight pruning is used, the number of weight $s_{j+1} \cdot s_j$ is reduced by the pruning factor $q_{\text{prune}}^{(j)}$. However, additional information must be stored in the memory to determine the positions of the remaining weights (see Section 5.6). Therefore, the size for storing a b_{weight} bit weight is increased by the factor $q_{\text{overhead}} \geq 1$. The resulting formula with pruning is:

$$t_{\text{mem}} = \frac{s_{j+1} \cdot s_j \cdot b_{\text{weight}} \cdot q_{\text{overhead}} \cdot \left(1 - q_{\text{prune}}^{(j)}\right) \cdot N}{T_{\text{mem}} \cdot n}$$

The output calculation and the weight transfers are running in parallel. Therefore, the maximum of both times determines the overall processing time t_{proc}:

$$t_{\text{proc}} = \max(t_{\text{calc}}, t_{\text{mem}})$$

It can be seen that pruning is a very efficient measure to increase the throughput of embedded neural network processing due to the fact that the weight transfers are reduced as well as the number of calculations. However, an accuracy reduction might be taken into account. On the other hand, batch processing has no influence on the overall accuracy while significantly reducing the number of weight transfers. However, the number of operations remain the same, and an increased processing latency has to be taken into account.

By comparing the reduction of operations and memory transfers through pruning, it can be seen that the number of calculations is reduced by the inverse of the pruning factor $1 - q_{\text{prune}}^{(j)}$, whereas the number of data to transfer is only reduced by $\left(1 - q_{\text{prune}}^{(j)}\right) \cdot q_{\text{overhead}}$. In comparison, batch processing reduces only the amount of data transfers. Therefore, both methods are very effective in order to increase the overall throughput.

The network architecture, the achieved pruning factor $q_{\text{prune}}^{(j)}$, and the size of each weight b_{weight} are determined and optimized in the learning phase to achieve the required accuracy. To optimize the overall throughput of the hardware architecture with the given and above mentioned parameters, the number of processing resources $m \cdot r$ can be maximized and an optimal batch size n_{opt} can be calculated. This

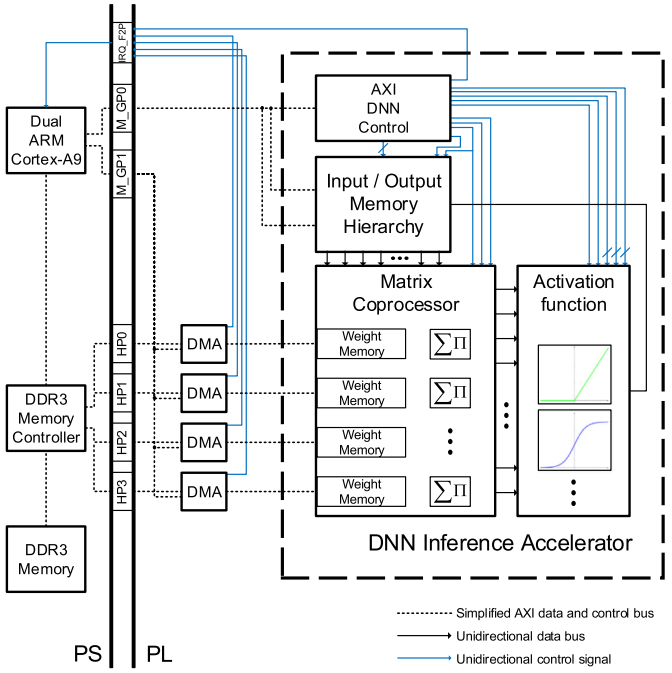

Fig. 4. Overview of our DNN accelerator with the Zynq processing system (PS) on the left and the custom accelerator inside the programmable logic (PL) on the right. The connecting PS-PL interfaces are shown in between. In addition, four DMA master peripherals are used for the weight transfer.

is achieved by setting $t_{mem} = t_{calc}$, i.e. neither the memory interface nor the MAC units have to wait for data or requests. This optimal batch size n_{opt} can be calculated with

$$n_{\text{opt}} \approx \frac{m \cdot r \cdot f_{pu} \cdot b_{\text{weight}} \cdot q_{\text{overhead}}}{T_{\text{mem}}}.$$

5. Architecture

We have implemented two architectures to demonstrate the batch processing and pruning approach on Xilinx's *Zynq-7000 All Programmable SoC* platform [25]. This SoC represents an affordable, low power device with a recent FPGA fabric that is suitable for various embedded systems. An overview visualizing the overall accelerator structure with generic components for both designs and all related Zynq peripherals is shown in Fig. 4.

FPGA-based coprocessors for the Zynq usually depend highly on the interfaces between the *processing system* (PS) and the *programmable logic* (PL) in order to achieve the highest transfer bandwidth. In our case, this is especially true for the DDR3 memory controller that resides inside the PS and is used to retrieve the network weights.

All major connections that cross the boundary of our actual DNN accelerator are indicated as dashed lines in Fig. 4. These buses pass several interconnects and controllers, both inside the PS and the PL, which are necessary for the communication but are omitted in the visualization in order to simplify the overview and to focus on the most important aspects.

In general, the software running on the ARM cores of the Zynq is used to configure and monitor both the control unit of the accelerator and all four DMA engines. It is also meant to transfer the network input and outputs.

The actual processing begins as soon as both the first inputs from the software and the first weights from a burst transfer of the DMA engines arrive. For this purpose, both accelerators share an overall similar architecture which is divided into four major IPs. However, depending on the concrete design, each of these IPs is differently implemented. Similarities and differences are detailed in the following:

5.1. Control unit

The first IP, called *AXI DNN Control*, is the control unit of the three remaining datapath IPs in Fig. 4. In addition, it stores metadata, like the dimension of the matrix operation, or certain runtime adjustable settings, like the type of the activation function (e.g., ReLU or sigmoid). It also monitors the current processing stage and is able to precisely inform the software side about requests and events like required data transfers. Furthermore, it stores additional design specific information (e.g., the batch size for the batch processing design).

5.2. Input / output memory hierarchy

Both accelerators have an internal memory hierarchy that is used to store input and output activations for the currently calculated layer. While the input for the first layer needs to be copied by the ARM cores, the inputs for the following layers are always outputs of previous layers and thus computed and stored inside the memory hierarchy. The flexibility to arbitrarily act as input or output requires a certain hierarchy of multiplexers, demultiplexers and multiple memory ports since the data must be accessible by the processing system and multiple compute units inside the programmable logic. Depending on the design, some degree of redundancy is required in order to avoid pipeline stalls. This is further explained in Section 5.6.

5.3. Matrix coprocessor

The IP with the largest resource utilization is the matrix coprocessor that computes the transfer function, i.e., the weighted sum of inputs $z_i^{(j)}$. This involves matrix-vector (pruning) or matrix-matrix (batch processing) operations that are mainly implemented with multiply-accumulate units (MACs) by using DSP slices. We use a fixed point data format, known as Q7.8, that consists of one sign bit, seven integer bits and eight fractional bits. Although there exist first results that use fewer bits for both weights and activations (e.g., between 1 and 8 bits) [29], 16 bits are, as of today, the most frequently used bit-width. For the DNN inference, this format is proven to be almost as accurate as single precision floating point weights [14,16,26], whereas weight encodings with very few bits (e.g., 1 or 2 bits) suffer from comparable low accuracy [27]. Note that multiplications use 16 bits, while the subsequent accumulation is done with 32 bits. This ensures that the input of the activation function is provided with a full precision of 32 bits (e.g., Q15.16).

5.4. Activation function

All activation functions in our designs are implemented using comparators and arithmetic operations. Due to the small number of cases in modern activation functions like ReLU, an efficient implementation using combinational logic is possible while occupying only few logic resources. More complex functions (e.g., sigmoid) are implemented using the piecewise linear approximation (PLAN) that was originally proposed by Amin et al. [30]. The desired function can be dynamically chosen by the *AXI DNN Control* which allows both accelerators to support different layer types. Older implementations also used precomputed activation function images stored in lookup-tables [12]. However, as explained in [8] theses tables occupy valuable memory resources and are less flexible considering a dynamic change of the actual function type.

5.5. Datapath throughput optimzation - batch processing

In order to efficiently process multiple input samples, the previously

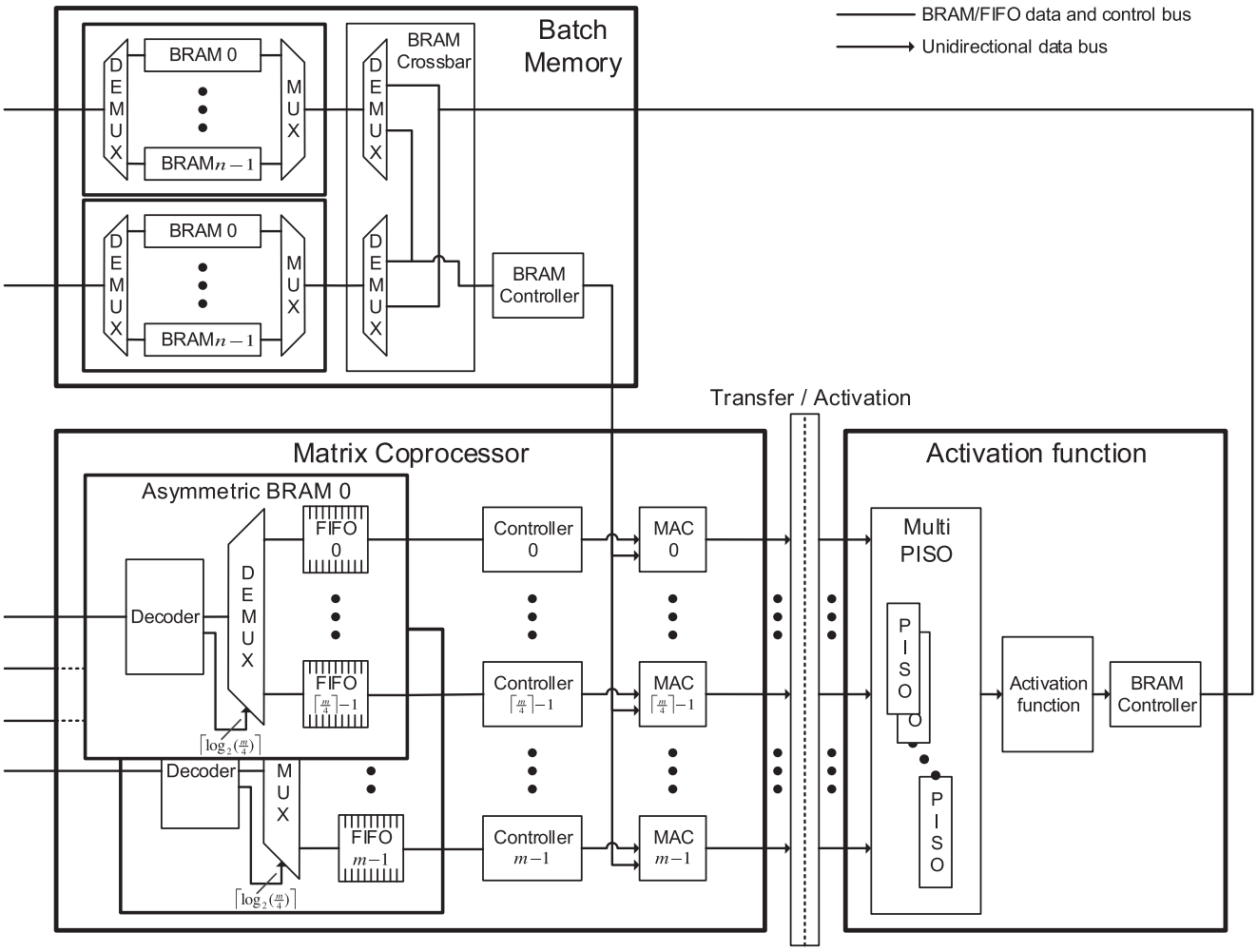

Fig. 5. Datapath for the batch processing of deep neural networks. The batch memory contains two dedicated memory hierarchies for the previous and the current computed layer. Each of the two memories contains n storage elements for the n processed samples. The crossbar can switch between the input and output role of a layer. The coprocessor and activation function implement the processing of $m \cdot n$ neurons before a software intervention is required.

discussed datapath components have to be adapted. Fig. 5 shows the conceptual mapping of an arbitrary batch size with up to n samples.

The memory hierarchy, here called *Batch Memory*, contains n BRAMs for both input and output activations. Due to the regular structure of the matrix-matrix operation in batch processing (cf. Section 4.4), the BRAM controller inside the batch memory can prefetch the correct input of the current section without stalling the pipeline and supply it to all m parallel MACs. Note that in this architecture, all m processing units, one for each neuron, have only a single MAC unit. Therefore, r is set in this case to $r = 1$. At the *same time*, the activations of the *previous* section can be written into the memory. The BRAM crossbar facilitates that each BRAM can play either the role of the input or the output, depending on the current processing state.

Using the batch memory hierarchy for the input activations and FIFOs for the corresponding weights, the *Matrix Coprocessor* calculates the transfer function for up to m neurons in parallel. The concrete number m is only restricted by the number of available DSP slices and BRAM resources. Often times the BRAMs are the limiting factor for the number of parallel processing units since at least one FIFO must be associated to one MAC unit in order to supply the weights. A FIFO stores up to one row $w_{i,0}^{(j)} \cdots w_{i,s_j-1}^{(j)}$ (the complete row if the previous layer is small enough) of the current weight matrix and is embedded in one of the four *asymmetric BRAMs* that are connected to the DMA engines.

For the final computation of a neuron, the results of the coprocessor are passed to the activation function. In case of batch processing, the complete design only requires one actual implementation of each function. A series of *Parallel In, Serial Out* registers is used to serialize the coprocessor outputs for a subsequent activation function. A more detailed description can also be found in [8].

Internally, all three datapath components of the batch processing design contain an extensive pipelining. Although these pipeline stages exist, Fig. 5 visualizes only one pipeline register between the coprocessor and the activation function. This stage is crucial for the batch processing since it allows a full decoupling of the transfer and activation function (i.e., both work in parallel using different samples).

Since we defined our section size to be $s_{j+1} \geq m$ and the coprocessor needs s_j clock cycles for all MAC operations of the section if $r = 1$, our activation function can take up to s_j cycles before the next coprocessor results are in the main pipeline stage. Furthermore, both the ReLU and the sigmoid function are implemented using one clock cycle ($c_a = 1$). Hence, our design of only one active activation function reduces the required FPGA logic resources considerably without any throughput declines. In general, the computation of the layer $j + 1$ (including the cycles for the activation function) across all n samples requires

$$\left\lceil \frac{s_{j+1}}{m} \right\rceil \cdot s_j \cdot n + m \cdot c_a$$

clock cycles since only the activations of the last section for sample n are not computed in parallel ($m \cdot c_a$). Moreover, for this term $m \cdot c_a \ll s_{j+1} \cdot s_j$ holds true. Thus, the approximate time for calculating all results of layer $j + 1$ is

$$t_{calc} \approx \frac{s_{j+1} \cdot s_j \cdot n}{m \cdot f_{pu}},$$

which is the same as the formula in Section 4.4 with $N = n$, $q^{(j)}_{\text{prune}} = 0$, and $r = 1$.

5.6. Datapath throughput optimization - pruning

Compared to the batch processing design, where it is sufficient to transfer a sequence of weights and the dimension of the matrix operation, pruning requires additional metadata that gives information about the actual position of a weight $w^{(j)}_{i,k}$ within the matrix $W^{(j)}$ as stated in Section 4.3. We use a format similar to [19] that represents individual rows of the sparse weight matrices using tuples of (w_l, z_{w_l}) entries, with $l = 0 \cdots \left(1 - q^{(j)}_{\text{prune},k}\right) \cdot s_j - 1$. Here, w_l encodes a remaining weight after pruning and z_{w_l} denotes the number of preceding zeros that come before w_l in the corresponding row. The number of remaining weights after pruning is $s_j \cdot \left(1 - q^{(j)}_{\text{prune},k}\right)$, where $q^{(j)}_{\text{prune},k}$ is the pruning factor of row k of the weight matrix $W^{(j)}$. The overall pruning factor $q^{(j)}_{\text{prune}}$ of the weight matrix $W^{(j)}$ can be calculated with

$$q^{(j)}_{\text{prune}} = \frac{1}{s_{j+1}} \cdot \sum_{k=0}^{s_{j+1}-1} q^{(j)}_{\text{prune},k}.$$

Opposed to [19], we do not separate the weights and zeros into two 1-dimensional arrays and store them in on-chip tables, but rather pack a certain number r of consecutive (w_l, z_{w_l}) tuples into one *data word* (cf. [31]). In our architecture we use $r = 3$ tuples, encode w_l with the Q7.8 format (the same as in the batch processing approach), and represent z_{w_l} as an unsigned integer with 5 bits. Using these parameters, a row

$$(0,\ -1.5,\ 0,\ 0,\ +0.3,\ -0.17,\ 0,\ 0,\ 0,\ +1.1,\ 0,\ 0,\ -0.2,\ 0,\ +0.1,\ \cdots)$$

is encoded into the following sequence of 64 bit data words

Note that this encoding uses only 63 bit from the 64 bit available data word. The advantage is that the data is memory aligned to the 64 bit border which eases the memory access. The corresponding overhead per weight compared to non-pruning implementations is $q_{\text{overhead}} = 64 \text{ bit}/(3 \times 16 \text{ bit}) = 1.33$.

Compared to other sparse matrix encodings that, for example, use separate vectors for the absolute row and column pointers [20], this format works well for streaming architectures since it directly combines both the weight and its relative position in one stream. This means that it does not require synchronization for, e.g., weight and multiple index streams. Since the structure of pruned weight matrices is not as homogeneous as their dense counterparts, the datapath of a corresponding streaming architecture must be designed to handle sparse matrices in order to avoid pipeline stalls. A datapath adaptation that supports the discussed format is depicted in Fig. 6.

Where the fully-connected structure assured that each input activation $a^{(j)}_k$ is needed for the computation of each neuron $a^{(j+1)}_i$, in pruned weight matrices many input activations can be skipped due to corresponding zero-weights in the layer. Hence, in the batch processing datapath an input activation $a^{(j)}_k$ is supplied to all m parallel MAC units, whereas in the pruning datapath the coprocessor needs to calculate the address of the input activation $a^{(j)}_k$ for the current weight. This input address is potentially different for every row, which makes a parallel distribution of the inputs impractical. Therefore, each of the m parallel sparse row coprocessors has it own I/O memory unit. This means that the I/O memory and the coprocessors are replicated m times. Each of the m I/O memories is addressed individually. To calculate the address in order to access the corresponding input activation $a^{(j)}_k$, the following formula can be used:

$$\text{address}_l = l + \sum_{k=0}^{l-1} z_{w_k}$$

The *offset calculation* IP computes these addresses for all r weights iteratively using the previously computed and stored offset o_{reg}, the number of non-zero weights before w_l and the zero fields z_{w_l} from the pipeline word:

$$\text{address}_i = o_{\text{reg}} + i + \sum_{k=0}^{i} z_{w_k},$$

where $i = 0 \cdots r - 1$. Depending on the number of tuples r, this means for the hardware design either a longer combinational path with r subsequent adders or otherwise adders with up to $r + 1$ inputs. Since r in our design is sufficiently small ($r = 3$), we use the latter.

Having computed the addresses, the coprocessor can multiply the weights and retrieve input activations and subsequently accumulate the partial sums. However, in order to retrieve the weights in parallel and avoid multiple cycles for a sequential fetching of the individual activations, the input memory needs r read ports. Given that RAM resources in current FPGA technologies usually do not provide more than two memory ports [32], the *I/O memory* inside the pruning datapath stores both input and output activations in r redundant BRAM copies. This provides at any time the required r memory ports. Compared to the batch processing datapath, the I/O memories in the pruning datapath only store one sample. When m neurons should be computed in parallel, this redundancy is even increased to $m \cdot r$ copies since each of the m coprocessors needs r individual read ports. If the calculated address_i surpasses the stored number of inputs s_j, the calculation of the current transfer function $z^{(j+1)}_i$ is finalized, the result is handed over to the activation function, and the corresponding processing unit starts calculating the following transfer function $z^{(j+1)}_{i+m}$. After the activation function, a merger IP (not depicted in Fig. 6) distributes the computed output activations of the m neurons to *all* I/O memories (second port of the BRAM crossbar). This requires a round-robin multiplexing scheme of the involved FIFOs after the activation function. Opposed to the batch processing datapath (cf. Fig. 5), we decided to use m hardware activation functions since the number of accumulations depends now on the percentage of the pruned parameters which might differ on a case-by-case basis.

6. Experimental results

To evaluate and verify the so far discussed concepts, we have implemented both presented accelerators on an embedded platform and compared them with different configurations against miscellaneous software platforms. In this section, we experimentally determine parameters like the best performing batch size n and show how beforehand chosen parameters perform. Furthermore, we show on the target hardware what performance gains are to be expected when both concepts are correspondingly implemented.

We chose the *Zynq Evaluation and Development Board* [33], short *ZedBoard*, for the implementation of our designs. It represents a typical embedded SoC and features a XC7020 device with Artix-7 FPGA fabric.

For the dedicated hardware support of different batch sizes we synthesized multiple bitstreams (a more detailed resource utilization is given in [8]) whereas the pruning design is only synthesized once with the parameters $m = 4$ and $r = 3$.

Each design uses two clock domains: the memory interface (e.g., Zynq high performance ports and DMAs) is clocked with 133 MHz and the remaining processing IPs use a 100 MHz clock (f_{pu}).

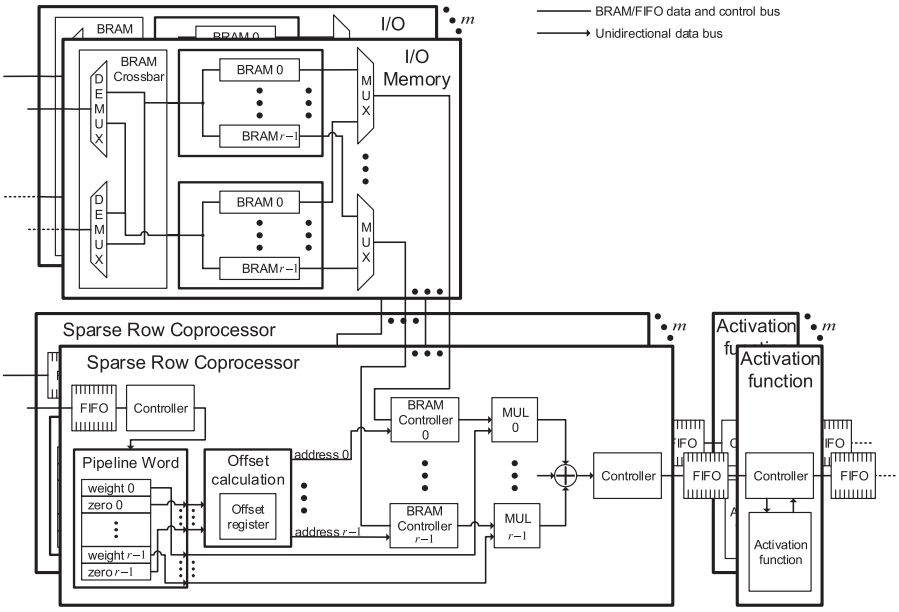

Fig. 6. Datapath for the computation of sparse rows in pruned DNNs. This example presumes a pipeline word with r tuples, each containing a weight and the number of zeros before it. In order to avoid delays when fetching the input activation that corresponds to a given weight, the BRAMs in the I/O memory are also duplicated r times, such that each multiplier has its own memory port. By combining m of these datapath instances, m neurons can be computed in parallel (i.e., m rows of the sparse matrix). In such cases, an IP that merges the activations of different rows must be connected with the I/O memories (indicated through the dashed lines).

6.1. Throughput evaluation

For a fair comparison of both hardware and software, we have trained different fully-connected neural network architectures with multiple real-world data set. As many before us, we use the famous *MNIST database of handwritten digits* [34] as the first benchmark. The data set contains 60,000 training and 10,000 test samples. A sample represents in this case a digit between 0 and 9, and is given as a grayscale image with a resolution of 28 × 28 pixels. In addition, we have also performed all tests with a second benchmark that deals with the subject of recognizing human activities (*HAR*) of daily living through smartphone sensors [35]. For this purpose, a person (who is wearing the smartphone) performed one of six activities (walking, walking upstairs, walking downstairs, sitting, standing, and laying). One sample of the data set is a 561-feature vector of time and frequency variables from different smartphone sensors (accelerometer, gyroscope, etc.). A use case could be tracking of sport activities or, in a batch scenario, complete sequences of motions. The data set is divided into 7352 training and 2947 test samples.

In our evaluation, all hardware candidates compete against a software implementation that we have tested on an embedded (i.e., the ZedBoard without FPGA use), a notebook (DELL Latitude E7250 Ultrabook) and, a desktop machine. A more detailed hardware specification of all three platforms is given in Table 1. Xilinx's bare-metal layer is used for the ZedBoard whereas both the notebook and the desktop machine use Linux-based operating systems. By default, bare-metal uses only one core for the software execution.

Table 1
Detailed hardware specification of the three machines used for the software DNN processing.

Machine	ARM Cortex-A9	Intel Core i7-5600U	Intel Core i7-4790
CPU clock freq. (MHz)	667	2600–3200	3600–4000
Cores (Threads)	2 (2)	2 (4)	4 (8)
L1 cache size (KB)	32	128	256
L2 cache size (KB)	512	512	1024
L3 cache size (KB)	—	4096	8192
Total RAM (MB)	512	8192	16384
Dual channel used	no	no	yes
DDR3 controller peak bandwidth (GB/s)	4.2	12.8	25.6

Furthermore, all presented processors feature some variant of a vector extension to accelerate floating-point intensive calculations through parallelism on instruction level. For the ARM Cortex-A9 this extension is called NEON [36] whereas both Intel CPUs can use SSE and AVX for this purpose [37]. In order to get the best runtime result on *all* presented platforms, we use the *BLAS* [38] library for the software inference of the DNNs. The library is individually configured and compiled for each of the used processors. Note that the software is using 32-bit single-precision floating point numbers, whereas our hardware design uses the described Q7.8 fixed point format. The throughput results for the DNN inference on all software and hardware platforms are depicted in Table 2.

In order to measure the inference times on the individual platforms, we query the hardware performance counters of the processors before and after the actual computation of the DNNs. Similarly, for our hardware accelerator, we use its control software (running on one ARM core) and read the cycle count before triggering the computation and after the computation is done. In addition, the shown results are also averaged over the complete test set of the used benchmark. The inference times (i.e., time difference) are then given in milliseconds (ms) per samples.

Besides different software and hardware platforms, we have also tested multiple neural network architectures which are taken or inspired from current research in the field. For example, the smaller network for MNIST was proposed in [39] while the larger one is an artificially extended version of that architecture with four additional hidden layers.

The best results for both hardware designs and all software runs are highlighted. As visible, a pipeline with a batch size of 16 samples delivers the fastest ratio of input data and processing on the XC7020 target device. The optimal calculated batch size n_{opt} for the presented design is 12.66, assuming a constant number of $m = 114$ processing units clocked with $f_{pu} = 100$ MHz and the used Q7.8 fixed point format. On the software side, we see the fastest inference for the desktop machine with a utilization of 4 threads and dual channel (DC) memory. The results of the ARM core are significantly slower than all other platforms. A carefully written software implementation with *fixed point* numbers (i.e., only 16 bits per weight) and the NEON extension could theoretically be about four times faster. However, even then, the results would be multiple times slower than the hardware candidate with batch size 1 and more than an order of magnitude slower than most batch processing configurations. On both the mobile and desktop

Table 2
Throughput comparison of our hardware-based batch processing (multiple configurations of hardware batch sizes), our hardware design with pruning support, and software inference on three different systems. Execution times are averaged over the size of the used test set and given in milliseconds (ms) per sample.

Device	Configuration	MNIST[a]		HAR[b]	
		4-layer netw. 1,275,200 Parameters	8-layer netw. 3,835,200 Parameters	4-layer netw. 1,035,000 Parameters	6-layer netw. 5,473,800 Parameters
Hardware-based batch processing					
Batch size 1	114 MACs	1.543	4.496	1.3817	5.337
Batch size 2	114 MACs	0.881	2.520	0.7738	2.989
Batch size 4	114 MACs	0.540	1.505	0.463	1.792
Batch size 8	106 MACs	0.375	1.012	0.313	1.250
Batch size 16	90 MACs	0.285	0.768	0.262	1.027
Batch size 32	58 MACs	0.318	0.914	0.287	1.203
Hardware-based pruning					
	Pruning factor	0.72	0.78	0.88	0.94
Pruning design	12 MACs	0.439	1.072	0.161	0.420
Software-based processing[c]					
ARM Cortex-A9	#Threads: 1	16.151	48.603	13.120	70.240
Intel core i7-5600U	#Threads: 1	0.285	1.603	0.223	2.246
	#Threads: 2	0.221	1.555	0.144	2.220
	#Threads: 4	0.247	1.591	0.182	2.417
Intel core i7-4790	#Threads: 1	0.118	0.917	0.114	1.406
	#Threads: 4	0.057	0.569	0.045	1.205
	#Threads: 8	0.065	0.687	0.055	1.491

[a] Network architectures: $784 \times 800 \times 800 \times 10$ and $784 \times 800 \times 800 \times 800 \times 800 \times 800 \times 800 \times 10$.
[b] Network architectures: $561 \times 1200 \times 300 \times 6$ and $561 \times 2000 \times 1500 \times 750 \times 300 \times 6$.
[c] Software calculations are performed using the *IEEE 754 floating point single precision* format and using *BLAS*. The i7-4790 utilizes dual channel memory whereas the others only use single channel.

CPU, the execution times depend mostly on the network size and, more precisely, on the matrix sizes of the individual layers. While the matrices of both 4-layer networks fit completely into the CPU caches and thus enable a slightly faster execution times, the tables are turned for matrices of the deep learning era. For example, the 6-layer HAR network with a 2000×1500 matrix represents such a typical fully-connected layer. Here, the hardware, despite its five times slower memory interface, clearly outperforms all software implementations.

As expected, the results of the pruning design are highly dependent on the actual pruning factor q_{prune}. The MNIST results with a pruning factor below 80% are comparable to the performance of the batch processing design with batch size $n = 8$. However, in the HAR benchmark where more than 90% of the parameters were pruned, the performance clearly surpasses the best batch processing results. Due to the very limited amount of 4 high performance ports on the Zynq, our design utilizes only $m = 4$ coprocessors. This results in a total utilization of only 12 MACs.

Furthermore, we compared our approach with a related FPGA-based neural network accelerator. A fair and direct comparison is only possible with approaches that supply results for fully-connected DNNs or RNNs (RNNs have only slightly more weights due to neuron feedback connections). Apart from our presented batch processing scheme, accelerators for fully-connected layers can in general only use a weight once per network computation. Instead, CNNs are able to reuse weights due to a different neuron connection pattern in the convolutional layers. Hence, they naturally achieve higher GOps/s due to a lower memory throughput requirement in direct comparisons with fully-connected layers. However, when considering *only* fully-connected layers the presented batch processing scheme clearly outperforms related work like, for example, a recent RNN approach on the ZedBoard [18]. The authors claim an overall throughput of 388.8 MOps/s. With our approach and by using batch size $n = 16$, we reach a throughput of 4.48 GOps/s and 5.00 GOps/s, respectively (only counting MAC operations). Although they are using less resources, our approach has a 6 times better throughput per DSP slice and 3 times better throughput per LUT and FF. The architecture using the pruning approach has only 0.8 GOps/s due to removed weights and operations. However, compared with non-pruned approaches, this is equivalent to 2.91 GOps/s and 3.58 GOps/s, respectively.

6.2. Energy efficiency

Even though our approach outperforms almost all of the x86-based software configurations or has at least a comparable throughput, the real benefit is evident when comparing the energy efficiency. For determining the energy consumption, we measured the system power for processing the 8-layer neural network, introduced in Section 6.1, and the idle power for all platforms (see Table 3). The overall power consumption on the ZedBoard is evaluated by measuring the average input voltage and the voltage drop on a shunt resistor. Whereas, the average power of the x86-based systems is measured on the primary side of the power supply with an ampere and volt meter. Besides the idle and processing power, the energy consumption with (Overall Energy) and without (Dynamic Energy) the idle power consumption is shown in Table 3.

Comparing our best performing hardware configuration of batch size $n = 16$ with pure software approaches, an overall energy efficiency improvement of almost factor 10 and more than factor 12 for the dynamic energy can be achieved. In the latency measurements, the i7-5600U is the nearest competitor.

Table 3
Energy consumption comparison of our hardware designs and three processors (network: MNIST 8-layer).

Device	Configuration	Power (W)	Overall Energy (mJ)	Dynamic Energy (mJ)
ZedBoard	idle	2.4	—	—
	HW batch ($n = 16$)	4.4	3.8	1.5
	HW pruning ($m = 4$)	4.1	4.4	1.8
	SW BLAS	3.8	184.7	68.0
Intel core i7-5600U	idle	8.9	—	—
	#Threads: 1	20.7	33.2	18.9
	#Threads: 2	22.6	35.1	21.3
	#Threads: 4	24.9	39.6	25.5
Intel core i7-4790	idle	41.4	—	—
	#Threads: 1	65.8	63.9	22.4
	#Threads: 4	82.3	46.8	23.3
	#Threads: 8	81.8	56.2	27.8

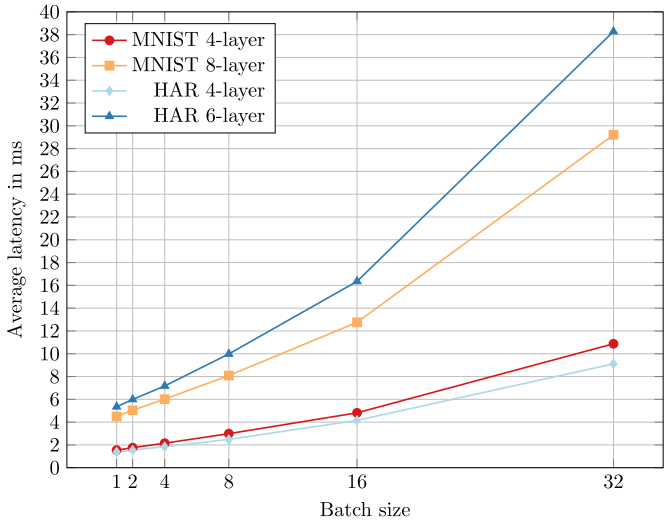

Fig. 7. Latency analysis for different batch sizes and network architectures. Latency is given in milliseconds and averaged over the test set of the network.

Compared to a competing LSTM design [21], our pruning approach is about factor 1.8 more energy efficient using their network with 3,248,128 weights, their pruning factor of $q_{\text{prune}} = 0.888$, and our theoretical throughput estimation of Section 4.4 (1.9 mJ for our pruning approach and 3.4 mJ for their approach).

6.3. Batch processing latency evaluation

As mentioned earlier, the presented batch processing approach represents a trade-off between throughput and latency. Fig. 7 compares the averaged latency of samples with the configured batch size.

For all of the investigated networks, a batch size of 8 samples results in approximately the doubled latency compared to regular processing with only 1 sample. The best throughput configuration of batch size 16 yields approximately the tripled latency in comparison with regular processing.

6.4. Accuracy evaluation

Since our batch processing accelerator utilizes all weights and the same Q7.8 fixed point data format as most related works [14,16,26,40], we obtain in this case similar results concerning the accuracy. A more detailed description, that also takes a different ratio of integer and fractional bits into account, can be found in [8]. The objective for the training with pruning was a maximum accuracy deviation of 1.5% in correctly predicted samples. All networks discussed in the throughput evaluation (i.e., Section 6.1) meet this objective and deliver an accuracy very similar to their non-pruned counterparts (most deviate less than 0.5%). A detailed comparison of accuracy and pruning percentage is shown in Table 4.

Table 4
Accuracy evaluation in percentage of correctly predicted test set samples depending on the overall pruning factor q_{prune} of the network.

	MNIST[a]		HAR[b]	
Number of parameters	4-layer network 1,275,200 Parameters	8-layer network 3,835,200 Parameters	4-layer network 1,035,000 Parameters	6-layer network 5,473,800 Parameters
Best non-pruned accuracy	98.3		95.9	
Pruning factor	0.72	0.78	0.88	0.94
Accuracy	98.27	97.62	94.14	95.72

[a] Network architectures: $784 \times 800 \times 800 \times 10$ and $784 \times 800 \times 800 \times 800 \times 800 \times 800 \times 800 \times 10$.
[b] Network architectures: $561 \times 1200 \times 300 \times 6$ and $561 \times 2000 \times 1500 \times 750 \times 300 \times 6$.

7. Conclusions and future work

In this paper, we present two architectures for an FPGAs-based embedded SoC that are able to accelerate the inference of previously learned fully-connected deep neural networks. Both designs mitigate a slow access to the external memory on such SoCs by either reusing weights across multiple input samples (batch processing) or by pruning weights that do not affect the accuracy. Our comparison shows that these orthogonal techniques are both able to substantially increase the throughput of DNN accelerators. However, while batch processing does not affect the DNN accuracy, it may only be used in scenarios where an increased latency is tolerable. On the contrary, pruning can possibly reduce the accuracy but also even surpass the batch processing throughput results. In general, we were able to prune at least over 70% of parameters without noticeable accuracy drops and, for example, process 8 input samples in parallel with just a doubled latency. Additionally, each presented technique outperforms fully-featured x86-based systems once the size of the weight matrices is larger than the available cache. Thereby, each technique results in an energy-efficiency that is more than an order of magnitude better. Similarly, while large FPGAs outperform our design in terms of pure GOps/s, our design implemented on the presented embedded FPGA is almost factor two more energy-efficient.

Future works on this topic might further increase the throughput by combining both techniques into one datapath. The theoretical results, calculated from the formulas in the throughput discussion in Section 4.4, show that such a combination would substantially increase the throughput. However, one problem might be the used memory resources. Both approaches need a high amount of additional on-chip memories which are scarce on small embedded devices. Nevertheless, an envisaged design with $m = 6$, $r = 3$, and $n = 3$ would be feasible on the used Zynq and would have an expected inference time of the 6-layer HAR of 186 µs. This would be over 6 times faster than our fastest used x86 processor system.

References

[1] K. Simonyan, A. Zisserman, Very deep convolutional networks for large-scale image recognition, CoRR (2014) arXiv:1409.1556.
[2] T.N. Sainath, B. Kingsbury, B. Ramabhadran, P. Fousek, P. Novak, A.-r. Mohamed, Making deep belief networks effective for large vocabulary continuous speech

recognition, Proc. ASRU (2011).
[3] S. Han, H. Mao, W.J. Dally, Deep compression: compressing deep neural network with pruning, trained quantization and Huffman coding, CoRR (2015) arXiv:1510.00149.
[4] Nvidia drive px. http://www.nvidia.com/object/drive-px.html.
[5] D. Amodei, R. Anubhai, E. Battenberg, C. Case, J. Casper, et al., Deep speech 2: end-to-end speech recognition in english and mandarin, CoRR (2015) arXiv:1512.02595.
[6] Y. LeCun, J.S. Denker, S. Solla, R.E. Howard, L.D. Jackel, Optimal brain damage, in: D. Touretzky (Ed.), Advances in Neural Information Processing Systems (NIPS 1989), 2 Morgan Kaufman, Denver, CO, 1990.
[7] Y. LeCun, L. Bottou, G. Orr, K. Muller, Efficient backprop, in: G. Orr, M. K. (Eds.), Neural Networks: Tricks of the Trade, Springer, 1998.
[8] T. Posewsky, D. Ziener, Efficient deep neural network acceleration through FPGA-based batch processing, Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig 2016), Cancun, Mexico, (2016).
[9] T. Posewsky, D. Ziener, A flexible FPGA-based inference architecture for pruned deep neural networks, Proceedings of the International Conference on Architecture of Computing Systems, (2018).
[10] F.M. Dias, A. Antunes, A.M. Mota, Artificial neural networks: a review of commercial hardware, Eng. Appl. Artif. Intell. 17 (8) (2004) 945–952, http://dx.doi.org/10.1016/j.engappai.2004.08.011.
[11] J. Misra, I. Saha, Artificial neural networks in hardware: a survey of two decades of progress, Neurocomputing 74 (1–3) (2010) 239–255, http://dx.doi.org/10.1016/j.neucom.2010.03.021. Artificial Brains
[12] M. Pietras, Hardware conversion of neural networks simulation models for neural processing accelerator implemented as FPGA-based SoC, Field Programmable Logic and Applications (FPL), 2014 24th International Conference on, (2014), pp. 1–4, http://dx.doi.org/10.1109/FPL.2014.6927383.
[13] J. Schmidhuber, Deep learning in neural networks: an overview, CoRR (2014) arXiv:1404.7828.
[14] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, Y. LeCun, NeuFlow: a runtime-reconfigurable dataflow processor for vision, Proceedings of Embedded Computer Vision Workshop (ECVW'11), (2011). (Invited paper)
[15] C. Farabet, Y. LeCun, K. Kavukcuoglu, E. Culurciello, B. Martini, P. Akselrod, S. Talay, Large-scale FPGA-based convolutional networks, in: R. Bekkerman, M. Bilenko, J. Langford (Eds.), Scaling up Machine Learning: Parallel and Distributed Approaches, Cambridge University Press, 2011.
[16] V. Gokhale, J. Jin, A. Dundar, B. Martini, E. Culurciello, A 240 G-ops/s mobile coprocessor for deep neural networks, IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), (2014), pp. 696–701, http://dx.doi.org/10.1109/CVPRW.2014.106.
[17] J. Jin, V. Gokhale, A. Dundar, B. Krishnamurthy, B. Martini, E. Culurciello, An efficient implementation of deep convolutional neural networks on a mobile coprocessor, IEEE 57th International Midwest Symposium on Circuits and Systems (MWSCAS), IEEE, 2014, pp. 133–136.
[18] A.X.M. Chang, B. Martini, E. Culurciello, Recurrent neural networks hardware implementation on FPGA, arXiv preprint, arXiv:1511.05552 (2015).
[19] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M.A. Horowitz, W.J. Dally, EIE: efficient inference engine on compressed deep neural network, CoRR (2016) arXiv:1602.01528.
[20] R.W. Vuduc, Automatic Performance Tuning of Sparse Matrix Kernels, University of California, Berkeley, 2003 Ph.D. thesis.
[21] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, H. Yang, W.J. Dally, ESE: efficient speech recognition engine with compressed LSTM on FPGA, CoRR (2016) arXiv:1612.00694.
[22] V. Nair, G.E. Hinton, Rectified linear units improve restricted Boltzmann machines, Proceedings of the 27th International Conference on Machine Learning (ICML-10), (2010), pp. 807–814.
[23] D. Clevert, T. Unterthiner, S. Hochreiter, Fast and accurate deep network learning by exponential linear units (ELUs), CoRR (2015) arXiv:1511.07289.
[24] D.C. Ciresan, U. Meier, L.M. Gambardella, J. Schmidhuber, Deep big simple neural nets excel on handwritten digit recognition, CoRR (2010) arXiv:1003.0358.
[25] Zynq-7000 All Programmable SoC Technical Reference Manual, v1.10, Xilinx Inc., 2015. http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf.
[26] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, O. Temam, DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning, Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, ACM, New York, NY, USA, 2014, pp. 269–284, http://dx.doi.org/10.1145/2541940.2541967.
[27] Y. Umuroglu, N.J. Fraser, G. Gambardella, M. Blott, P.H.W. Leong, M. Jahre, K.A. Vissers, FINN: a framework for fast, scalable binarized neural network inference, CoRR (2016) arXiv:1612.07119.
[28] A.C. McKellar, E.G. Coffman Jr., Organizing matrices and matrix operations for paged memory systems, Commun. ACM 12 (3) (1969) 153–165, http://dx.doi.org/10.1145/362875.362879.
[29] M. Courbariaux, Y. Bengio, BinaryNet: training deep neural networks with weights and activations constrained to +1 or −1, CoRR (2016) arXiv:1602.02830.
[30] H. Amin, K. Curtis, B. Hayes-Gill, Piecewise linear approximation applied to non-linear function of a neural network, IEEE Proc. Circuits Devices Syst. 144 (6) (1997) 313–317, http://dx.doi.org/10.1049/ip-cds:19971587.
[31] Designing Protocol Processing Systems with Vivado High-Level Synthesis, v1.0.1, Xilinx Inc., 2014. https://www.xilinx.com/support/documentation/application_notes/xapp1209-designing-protocol-processing-systems-hls.pdf.
[32] 7 Series FPGAs Memory Resources, v1.12, Xilinx Inc., 2016. http://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf.
[33] ZedBoard Hardware User's Guide, v2.2 edition, Avnet Inc., 2014. http://zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf.
[34] Y. LeCun, C. Cortes, C.J. Burges, MNIST handwritten digit database, 2014, (http://www.yann.lecun.com/exdb/mnist/).
[35] D. Anguita, A. Ghio, L. Oneto, X. Parra, J.L. Reyes-Ortiz, A public domain dataset for human activity recognition using smartphones, 21th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning, ESANN 2013, (2013).
[36] ARM Holdings PLC Cortex-A9 NEON Media Processing Engine Technical Reference Manual, http://www.infocenter.arm.com/help/topic/com.arm.doc.ddi0409g/DDI0409G_cortex_a9_neon_mpe_r3p0_trm.pdf, r3p0 edition, 2011.
[37] N. Firasta, M. Buxton, P. Jinbo, K. Nasri, S. Kuo, Intel AVX: new frontiers in performance improvements and energy efficiency, Intel White Paper, (2008).
[38] Z. Xianyi, et al., OpenBLAS, 2011, (http://www.openblas.net). Accessed: 2016-03-02.
[39] G. Hinton, O. Vinyals, J. Dean, Distilling the knowledge in a neural network, ArXiv e-prints (2015).
[40] S. Cadambi, I. Durdanovic, V. Jakkula, M. Sankaradass, E. Cosatto, S. Chakradhar, H.P. Graf, A massively parallel FPGA-based coprocessor for support vector machines, Annual IEEE Symposium on Field-Programmable Custom Computing Machines, (2009), pp. 115–122 http://www.doi.ieeecomputersociety.org/10.1109/FCCM.2009.34.

Thorbjörn Posewsky received his bachelor's and master's degree in Computer Science from the University of Paderborn. In 2016 he joined the Embedded Systems institute at Hamburg University of Technology (TUHH) as a research assistant. His main research interests include FPGA accelerators, hardware/software codesign, machine learning, and computer vision.

Daniel Ziener is currently an associate professor in the CAES group (Computer Architectures for Embedded Systems) at the faculty EEMCS of the University of Twente, Netherlands. From 2015 until 2017, he was a substitute professor for Cyber-Physical Systems at the Hamburg University of Technology, Germany. From 2010 to 2015, he had led the Reconfigurable Computing Group of the Chair of Hardware/Software Co-Design at Friedrich-Alexander University Erlangen-Nürnberg (FAU), Germany. His main research interests are the usage of partial dynamic reconfiguration of FPGAs, efficient usage of FPGA structures, design of signal processing FPGA cores, reliable and fault tolerant embedded systems, as well as security in FPGA-based systems. Daniel has (co-)authored more than 40 peer-reviewed publications, holds two patents, and serves as a program committee member of several international conferences (DATE, FPL, Reconfig, SPL) as well as a reviewer for several international journals. Daniel took his university entrance qualification in 1998. He received his diploma degree (Dipl.-Ing. (FH)) in Electrical Engineering from University of Applied Science Aschaffenburg, Germany, in August 2002. Beside his studies, he gained industrial research experience during an internship at the IBM Germany Development Labs in Böblingen. From 2003 to 2009 he worked for the Fraunhofer Institute of Integrated Circuits (IIS) in Erlangen, Germany as a research staff in the electronic imaging department. Furthermore, in 2003 he joined the Chair of Hardware-Software-Co-Design at the University of Erlangen-Nuremberg, Germany, headed by Prof. Jürgen Teich as Ph.D. student. In 2010 he received his Ph.D. degree (Dr.-Ing.) and in 2017 his habilitation (Dr.-Ing. habil.).