A Flexible FPGA-Based Inference Architecture for Pruned Deep Neural Networks

Thorbjörn Posewsky[1] and Daniel Ziener[2(✉)]

[1] Ibeo Automotive Systems GmbH, Hamburg, Germany
[2] Friedrich-Alexander University Erlangen-Nürnberg (FAU), Erlangen, Germany
daniel.ziener@fau.de

Abstract. In this paper, we present an architecture for embedded FPGA-based deep neural network inference which is able to handle pruned weight matrices. Pruning of weights and even entire neurons reduces the amount of data and calculations significantly, thus improving enormously the efficiency and performance of the neural network inference in embedded devices. By using an HLS approach, the architecture is easily extendable and highly configurable with a free choice of parameters like the number of MAC units or the used activation function. For large neural networks, our approach competes with at least comparable performance as state-of-the-art x86-based software implementations while only using 10% of the energy.

1 Introduction and Motivation

For more and more people, *Deep Neural Networks* (DNNs) have become a substantial part of their daily life. Applications like image classification [22] or speech recognition [20] 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 robotics or autonomous vehicles. Yet, these emerging areas have different power requirements and lack 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. Despite 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 [13]. 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 technique investigated in this work, now known as *pruning*, represents a form of DNN compression [13,17]. Pruning reduces the number of synaptic connections to adjacent neurons such that the overall amount of weights is reduced.

© Springer International Publishing AG, part of Springer Nature 2018
M. Berekovic et al. (Eds.): ARCS 2018, LNCS 10793, pp. 311–323, 2018.
https://doi.org/10.1007/978-3-319-77610-1_23

Most importantly, pruning is often able to eliminate a significant portion of these connections without or with just minor accuracy drops for, i.e., classification tasks. Due to the reduced amount of weights, less data needs to be transferred and less calculations are needed in the hardware. Correspondingly, accelerators are able to compute DNNs much faster. Currently, only a very limited number of previous works exist that consider dedicated hardware support for pruned DNNs [11,12].

As previously mentioned, deep learning can generally be used for many embedded computing applications. The inference efficiency of such embedded solutions plays a pivotal role and is highly dependent on the right hardware architecture for the application-specific neural network architecture. To support a wide area of different networks and, therefore, applications, we use an FPGA-based *high level synthesis* (HLS) [15] approach in order to design a very efficient hardware by rapidly exploring different design parameters, like the number of MAC units or different activation functions. In this paper, we show how a flexible streaming architecture for arbitrarily pruned DNNs can be designed as opposed to designs with partially or completely embedded parameters. 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: Sect. 2 gives an overview of related work. The concept and architecture of our accelerator is explained in Sects. 3 and 4, respectively. Section 5 continues with experimental results. Finally, Sect. 6 concludes the work and highlights future research directions.

2 Related Work

Recently, many accelerator designs for *Convolutional Neural Networks* (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 [21]. One example for such an accelerator is given in [10]. 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. However, while such a design and many others (e.g., [8]) are very effective for convolutional layers, their internal buffers and routing elements are not optimized for fully-connected or compressed networks.

An FPGA-based DNN inference architecture that specifically addresses fully-connected layers is presented in [19]. Additionally, the approach enables the reuse of previously transferred weight matrices across multiple input samples, which is referred to as batch processing. Both techniques, the one presented in this work and the one in [19], reduce data transfers for the inference of fully-connected DNNs significantly but are conceptually orthogonal.

A third important type of networks is known as *Recurrent Neural Network* (RNN) [21]. 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, which uses the same FPGA as this work, is shown in [3] and their results are accordingly compared to ours in Sect. 5.

The theoretical foundation for pruning and, thus, our accelerator was introduced by LeCun et al. in [17]. Originally, it was used to improve generalization and speed of learning in shallow network architectures. However, Han et al. [13] recently revived the technique for DNNs and were able to reduce the number of connections by a factor between 9x and 13x. A corresponding ASIC design with large on-chip memories for the remaining parameters after pruning and quantization (without Huffamn encoding) is given in [12]. As discussed later, our accelerator utilizes a similar format, presented in [24], 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 [11].

3 Concept

A typical neural network contains several layers $j = 1 \dots L$. A layer j itself consists of s_j neurons. Fully-connected layers in DNNs 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, e.g., $s_0 \times s_1 \times s_2$ for a $L = 3$ layer network. The synaptic strength of a connection is modeled through a scalar value $w_{i,k}^{(j)}$ called *weight* that represents the connection to the i-th neuron in layer $j+1$ from the k-th 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 and the outputs $a_k^{(j)}$ of connecting neurons in the layer j. The result of each neuron $a_i^{(j+1)}$ is computed by the following functions:

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

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 Units* (ReLU) [18] or variations of it [6].

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,

given 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 [5]. The network weights need approximately 22 MB if each weight is encoded using 16 bits. Compared to FPGA platforms like the Zynq, where even the largest device is limited to a total BRAM size of less than 3.3 MB [27] (i.e. 26.5 Mb ≈ 3.3 MB for the Z7100 device), a complete mapping with all neurons and weights directly onto the FPGA is no longer possible.

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. 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.

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. 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:

$$w_{i,k}^{(j)} < \delta \qquad \xrightarrow[\text{iterations}]{\text{following}} \qquad w_{i,k}^{(j)} := 0$$

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 accuracy drops [13].

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 to the compute units, or used in 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.

4 Architecture

We have implemented our design with support for pruned DNNs on Xilinx's *Zynq-7000 All Programmable SoC* platform [27] and using *Xilinx Vivado HLS*. The flexible HLS approach allows us to quickly elaborate the best performing architecture for a given neural network architecture. For example, the number of processing units m and the number of MAC units per unit r can be freely configured during design time. Moreover, the size and format of the weights and the kind of activation function can be easily exchanged. Furthermore, the design time is drastically reduced and our approach is easily extensible to support new

kinds of neurons, activation functions, or complete network architectures. An visualization of the overall accelerator structure and all related Zynq peripherals is shown in Fig. 1.

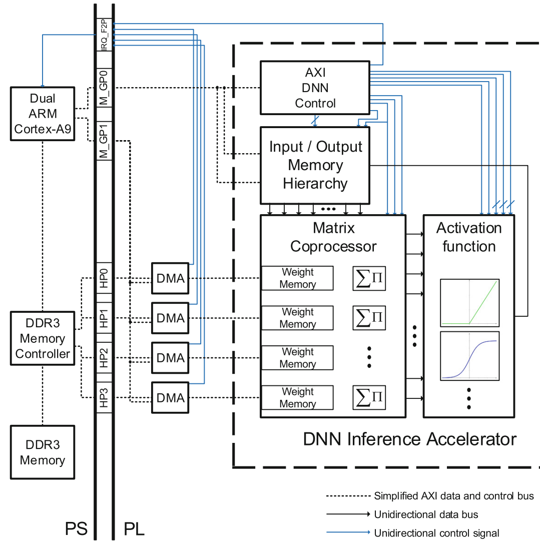


Fig. 1. Overview of our DNN accelerator with the Zynq processing system (PS) on the left and the custom accelerator inside the programmable logic (PL) on the right. The connecting PS-PL interfaces are shown in between. In addition, four DMA master peripherals are used for the weight transfer. All major connections that cross the boundary of our actual DNN accelerator are indicated as dashed lines.

The accelerator has an internal memory hierarchy that is used to store input and output activations for the currently calculated layer (controllable and accessible via software through the GP ports). 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 *Matrix Coprocessor* computes the transfer function, i.e., the weighted sum of inputs $z_i^{(j)}$. This involves matrix-vector 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) [7], 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 [4,9,10], whereas weight encodings with very few bits (e.g., 1 or 2 bits) suffer from comparable low accuracy [23]. 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 full precision (e.g., Q15.16).

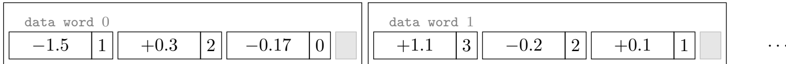
Compared to a design without pruning support where it is sufficient to transfer a sequence of weights and the dimension of the matrix operation, pruning requires additional metadata that gives information about the actual position of a weight $w_{i,k}^{(j)}$ within the matrix $W^{(j)}$. We use a format similar to [12] that represents individual rows of the sparse weight matrices using tuples of (w_l, z_{w_l}) entries, with $l = 0 \ldots (1 - q_{\text{prune},k}^{(j)}) \cdot s_j - 1$. 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 $s_j \cdot (1 - q_{\text{prune},k}^{(j)})$, where $q_{\text{prune},k}^{(j)}$ is the pruning factor of row k of the weight matrix $W^{(j)}$. The overall pruning factor $q_{\text{prune}}^{(j)}$ of the weight matrix $W^{(j)}$ can be calculated with

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

Opposed to [12], 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. [26]). In our architecture we use $r = 3$ tuples, encode w_l with the Q7.8 format, and represent z_{w_l} as an unsigned integer with 5 bits. Using these parameters, a row

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

is encoded into the following sequence of 64 bit data words



If z_{wl} would require more than 5 bits, e.g. more than 31 consecutive weights were pruned, we instead use multiple tuples with $(w_l, z_{wl}) = (0, 31)$ until the last tuple of the sequence holds the condition $z_{wl} < 31$. Note that the encoding of a data word uses only 63 bit from the available 64 bit. 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 [24], 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 design to handle sparse matrices in order to avoid pipeline stalls (see Fig. 2).

Therefore, the coprocessor needs to calculate the address of the input activation $a_k^{(j)}$ for the current weight. This input address is potentially different for

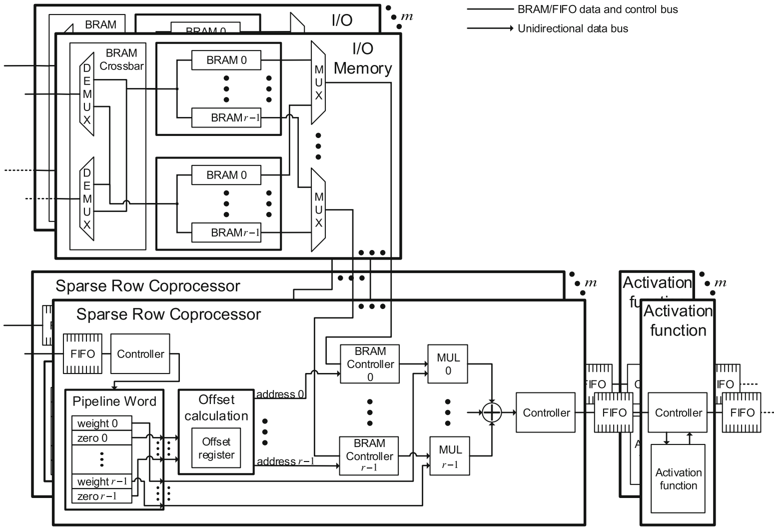


Fig. 2. Datapath for the computation of sparse rows in pruned DNNs. This example presumes a pipeline word with r tuples, each containing a weight and the number of zeros before it. In order to avoid delays when fetching the input activation that corresponds to a given weight, the BRAMs in the I/O memory 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).

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. The *offset calculation* IP computes theses 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:

$$\text{address}_i = o_{\text{reg}} + i + \sum_{k=0}^{i} z_{w_k}, \quad i = 0 \ldots r-1$$

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, the *I/O memory* stores both input and output activations in r redundant BRAM copies. 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 $z_i^{(j+1)}$ is finalized, the result is handed over to the activation function, and the corresponding processing unit starts calculating the following transfer function $z_{i+m}^{(j+1)}$. After the activation function, a merger IP (not depicted in Fig. 2) distributes the computed output activations of the m neurons to *all* I/O memories (second port of the BRAM crossbar).

5 Experimental Results

To evaluate and verify the so far discussed concept, we have implemented our accelerator on an embedded platform and compared them with different configurations against miscellaneous software platforms. We chose the *Zynq Evaluation and Development Board* [2], short *ZedBoard*, for the implementation of our designs. The 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}). Due to the limited amount of 4 high performance ports on the Zynq, our design utilizes only $m = 4$ coprocessors with $r = 3$ MAC units. This results in a total utilization of only 12 MACs. By using an HLS design flow, the design time was cut down to approximately 8 person weeks. In comparison, an earlier design (see [19]) with a similar complexity needed about 24 person weeks by using a standard RTL-based design flow (using VHDL). Furthermore, a substantial amount of the previously mentioned time for the HLS-based design was spent in creating a suitable testbench that is capable of loading arbitrary networks and transforming the weight matrices in the internal representation for the actual processing.

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 *MNIST database of handwritten digits* [16] as the first benchmark. 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 [1]. 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 [14] while the larger one is an artificially extended version of that architecture with four additional hidden layers.

In our evaluation, the hardware competes against a software implementation that we have tested on an embedded (i.e., the ZedBoard without FPGA use), a notebook and, a desktop machine. The notebook uses an Intel Core i7-5600U dual core processor with 2.6–3.2 GHz, 4096 KB L3 cache, and 8192 MB single channel DDR3 memory. The desktop CPU is an Intel Core i7-4790 quad core with 3.6–4.0 GHz, 8192 KB L3 cache, and 16384 MB dual channel DDR3 memory. The peak memory throughput is 12.8 GB/s for the laptop and 25.6 GB/s for the desktop system. The ZedBoard has only a memory throughput of 4.2 GB/s.

Furthermore, all presented processors feature some variant of a vector extension to accelerate floating-point intensive calculations through parallelism on instruction level. In order to get the best runtime result on *all* presented platforms, we use the *BLAS* [25] library for the software inference of the DNNs. 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. The throughput results for the DNN inference on all software implementations and our hardware platform are depicted in Table 1.

Table 1. Throughput comparison of 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 samples per milliseconds (ms). The best results for both hardware designs and all software runs are highlighted.

		MNIST [a]		HAR [b]	
Device	Configuration	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 processing					
	Pruning factor	0.72	0.78	0.88	0.94
HW 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.

On the software side, we see the fastest inference for the desktop machine with a utilization of 4 threads and dual channel memory. On both the mobile and desktop 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 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.

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). However, when considering *only* fully-connected layers, our approach clearly outperforms related work like, for example, a recent RNN approach on the ZedBoard [3]. The authors claim an overall throughput of 388.8 MOps/s. With our approach, we reach a throughput of 0.8 GOps/s (only counting MAC operations). However, compared with non-pruned approaches, this is equivalent to 3.83 MOps/1.07 ms = 3.58 GOps/s and 5.47 MOps/0.42 ms = 13.02 GOps/s, respectively (i.e., the non-pruned weight matrix is used as the number of operations, see Table 1).

Energy Efficiency: For determining the energy consumption, we measured the system power for processing the 8-layer neural network and the idle power for all platforms (see Table 2). 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) idle power consumption is shown in Table 2.

Table 2. Energy consumption comparison of our hardware design and three processors (network: MNIST 8-layer).

Device	Configuration	Power (W)	Overall Energy (mJ)	Dynamic Energy (mJ)
ZedBoard	idle	2.4	—	—
	HW ($m = 4$)	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

Comparing our hardware configuration with pure software approaches, an overall energy efficiency improvement of almost factor 10 can be achieved. Compared to a competing LSTM design [11], our pruning approach is about factor 1.8 more energy efficient using their network with 3248128 weights and their pruning factor of $q_{\text{prune}} = 0.888$ (1.9 mJ for our approach and 3.4 mJ for their approach).

Accuracy Evaluation: 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., Sect. 5) meet this objective and deliver an accuracy very similar to their non-pruned counterparts (most deviate less than 0.5%).

Table 3. 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 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
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 $561 \times 2000 \times 1500 \times 750 \times 300 \times 6$

A detailed comparison of accuracy and pruning percentage is shown in Table 3.

6 Conclusions

In this paper, we present a flexible architecture for an FPGA-based embedded SoC that is able to accelerate the inference of previously learned and arbitrary pruned fully-connected deep neural networks. This architecture enables the inference of today's huge networks on energy-constrained embedded devices. By using pruning, the size of external memory as well as the amount of data to be transferred can be significantly reduced which increases the energy efficiency and performance. An application and network-specific design can be easily achieved by using HLS in order to increase the abstraction level of the design entry. The resulting architecture has a comparable performance with state-of-the-art desktop and server processors for large networks. However, only a fraction of energy is needed which enables new applications for embedded systems, even on battery powered devices. Future works on this topic might further increase the throughput and energy efficiency by combining pruning with batch processing [19] into one architecture.

References

1. Anguita, D., Ghio, A., Oneto, L., Parra, X., Reyes-Ortiz, J.L.: A public domain dataset for human activity recognition using smartphones. In: 21th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning, ESANN 2013, April 2013
2. Avnet Inc.: ZedBoard Hardware User's Guide, v2.2 edn, January 2014
3. Chang, A.X.M., Martini, B., Culurciello, E.: Recurrent neural networks hardware implementation on FPGA. arXiv preprint arXiv:1511.05552 (2015)
4. Chen, T., Du, Z., Sun, N., Wang, J., Wu, C., Chen, Y., Temam, O.: Diannao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In: Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2014, pp. 269–284. ACM, New York (2014)

5. Ciresan, D.C., Meier, U., Gambardella, L.M., Schmidhuber, J.: Deep big simple neural nets excel on handwritten digit recognition. CoRR abs/1003.0358 (2010)
6. Clevert, D., Unterthiner, T., Hochreiter, S.: Fast and accurate deep network learning by Exponential Linear Units (ELUs). CoRR abs/1511.07289 (2015)
7. Courbariaux, M., Bengio, Y.: BinaryNet: Training deep neural networks with weights and activations constrained to +1 or −1. CoRR abs/1602.02830 (2016)
8. Farabet, C., LeCun, Y., Kavukcuoglu, K., Culurciello, E., Martini, B., Akselrod, P., Talay, S.: Large-scale FPGA-based convolutional networks. In: Bekkerman, R., Bilenko, M., Langford, J. (eds.) Scaling up Machine Learning: Parallel and Distributed Approaches. Cambridge University Press, Cambridge (2011)
9. Farabet, C., Martini, B., Corda, B., Akselrod, P., Culurciello, E., LeCun, Y.: Neuflow: a runtime-reconfigurable dataflow processor for vision. In: Proceedings of Embedded Computer Vision Workshop (ECVW 2011) (2011, invited paper)
10. Gokhale, V., Jin, J., Dundar, A., Martini, B., Culurciello, E.: A 240 G-ops/s mobile coprocessor for deep neural networks. In: IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), pp. 696–701, June 2014
11. Han, S., Kang, J., Mao, H., Hu, Y., Li, X., Li, Y., Xie, D., Luo, H., Yao, S., Wang, Y., Yang, H., Dally, W.J.: ESE: efficient speech recognition engine with compressed LSTM on FPGA. CoRR abs/1612.00694 (2016)
12. Han, S., Liu, X., Mao, H., Pu, J., Pedram, A., Horowitz, M.A., Dally, W.J.: EIE: efficient inference engine on compressed deep neural network. CoRR abs/1602.01528 (2016)
13. Han, S., Mao, H., Dally, W.J.: Deep compression: compressing deep neural network with pruning, trained quantization and Huffman coding. CoRR abs/1510.00149 (2015)
14. Hinton, G., Vinyals, O., Dean, J.: Distilling the knowledge in a neural network. ArXiv e-prints, March 2015
15. Koch, D., Hannig, F., Ziener, D. (eds.): FPGAs for Software Programmers. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-26408-0
16. LeCun, Y., Cortes, C., Burges, C.J.: MNIST handwritten digit database (2014). http://yann.lecun.com/exdb/mnist/
17. LeCun, Y., Denker, J.S., Solla, S., Howard, R.E., Jackel, L.D.: Optimal Brain Damage. In: Touretzky, D. (ed.) Advances in Neural Information Processing Systems (NIPS 1989), vol. 2. Morgan Kaufman, Denver (1990)
18. Nair, V., Hinton, G.E.: Rectified linear units improve restricted Boltzmann machines. In: Proceedings of the 27th International Conference on Machine Learning (ICML-2010), pp. 807–814 (2010)
19. Posewsky, T., Ziener, D.: Efficient deep neural network acceleration through FPGA-based batch processing. In: Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig), Cancun, Mexico, December 2016
20. Sainath, T.N., Kingsbury, B., Ramabhadran, B., Fousek, P., Novak, P., Mohamed, A.: Making deep belief networks effective for large vocabulary continuous speech recognition. In: Proceedings of the ASRU (2011)
21. Schmidhuber, J.: Deep learning in neural networks: an overview. CoRR abs/1404.7828 (2014)
22. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. CoRR abs/1409.1556 (2014)
23. Umuroglu, Y., Fraser, N.J., Gambardella, G., Blott, M., Leong, P.H.W., Jahre, M., Vissers, K.A.: FINN: a framework for fast, scalable binarized neural network inference. CoRR abs/1612.07119 (2016)

24. Vuduc, R.W.: Automatic performance tuning of sparse matrix kernels. Ph.D. thesis, University of California, Berkeley (2003)
25. Xianyi, Z., et al.: OpenBLAS, March 2011. http://www.openblas.net. Accessed 02 Mar 2016
26. Xilinx Inc.: Designing Protocol Processing Systems with Vivado High-Level Synthesis, v1.0.1 edn, August 2014
27. Xilinx Inc.: Zynq-7000 All Programmable SoC Overview, v1.9 edn, January 2016