Efficient Deep Neural Network Acceleration through FPGA-based Batch Processing

Thorbjörn Posewsky and Daniel Ziener
Institute of Embedded Systems
Hamburg University of Technology (TUHH)
21073 Hamburg, Germany
Email: {thorbjoern.posewsky, daniel.ziener}@tuhh.de

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 a novel architecture for processing previously learned and arbitrary deep neural networks on FPGA-based SoCs that is able to overcome these limitations. A key contribution of our approach, which we refer to as batch processing, achieves a mitigation of required weight matrix transfers from external memory by reusing weights across multiple input samples. This technique combined with a sophisticated pipelining and the usage of high performance interfaces accelerates the data processing compared to existing approaches on the same FPGA device by one order of magnitude. Furthermore, we achieve a comparable data throughput as a fully featured x86-based system at only a fraction of its energy consumption.

I. INTRODUCTION AND MOTIVATION

For more and more people, Deep Neural Networks (DNNs) have become a substantial part of their everyday life. Applications like image analysis (e.g., facial recognition) [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. 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.

Additionally, many applications use or even require multiple executions 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 [3]. Deploying speech recognition at scale (i.e. in data centers) is another example where a study [4] 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 DNNs executions are necessary or a small latency increase is tolerable.

While batch processing is a standard technique for an efficient DNN training [5] (called mini-batch processing in the context of stochastic gradient descent), it is so far not well investigated

- how this concept affects the design of hardware accelerators for the execution of DNN (forward-propagation),
- which gains in throughput can be achieved with a corresponding design,
- and what latency consequences are imposed by realizing this concept in dedicated hardware.

The contribution of this paper includes all the above mentioned points using an embedded FPGA with limited external memory bandwidth. We focus particularly on an efficient processing of fully-connected DNNs since the layers of these networks build the foundation for today's most successful network kinds.

The rest of this paper is organized as follows: Section II gives an overview of different network types together with specific hardware designs. The concept and architecture of our accelerator is explained in Section III and IV, respectively. Section V continues with experimental results for different hardware configurations, software platforms, and using several network architectures. Finally, Section VI concludes the work and highlights possible future work.

II. 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 [6] and [7]. 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 [8]. It is based on an array of so called Neural Processing Units (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 design a Time Division Multiplexing (TDM) processing scheme with a fast hardware-bases 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
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 [9]. 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 [10] and [11]. It relies on a two-dimensional grid of **Processing Tiles** (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 [12], the proposed design has scalability issues which is problematic for batch processing as shown in Section V. As a consequence, a CNN design with a linear array of processing elements (called collections) is shown in [13] and [12], 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) [9]. 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 network (i.e. that uses LSTMs) is shown in [14].

In general, one might ask how existing fine or coarse-grained reconfigurable architectures like CGRAs or TCPAs perform for the task of accelerating DNNs [15]. While their capabilities are for many applications beneficial, most of the functional elements and routing options of, for example, TCPAs are unnecessary for the execution of DNNs and occupy valuable resources.

### III. Concept

A typical neural network contains several layers \( j = 1 \ldots L \), 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 \( i \)-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. 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 \( j \).

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** \( \varphi \), 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 \( \varphi \) 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) [16] or variations of it [17]. 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 different functions at runtime.

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 [18]. The number of neurons is 8294 and the total size of the network weights is 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 2020 DSP slices and a total BRAM size of less than 3 MB [19, pp. 622], a complete mapping with all neurons and weights directly onto the FPGA is not possible.

Therefore, modern and deep neural networks need to be 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 \( m \leq s_j \), 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. Hereby, a section indicates the degree of parallelism and depends on the number of processing units in hardware. Since the network is fully-connected, the computation of layer \( j \) requires that all previous layers \( 1 \ldots j-1 \) are completely processed. Consequently, a
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. A straightforward section by section processing for just one sample has the drawback of exchanging these costly transferred weights for every new section. The input data (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. This huge demand of memory transfers turns the interface to the external memory into the major bottleneck for fully-connected layer processing. The main contribution of our 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 Figure 1.

![Conceptual batch processing](image)

Fig. 1. 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.

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 $1 \ldots n$ 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 [20]. However, as shown in the following Section IV, 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.

IV. ARCHITECTURE

We have implemented our proposed processing scheme on Xilinx’s Zynq-7000 All Programmable SoC platform [19]. This SoC represents an affordable, low power device with a recent FPGA fabric that is suitable for various embedded systems. An overview of our accelerator, together with all related Zynq peripherals, is shown in Figure 2.

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 Figure 2. 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 DMAs engines arrive. For this purpose, the complete DNN accelerator is divided into four major IP cores that are described in the following.

![Overview of DNN accelerator](image)

Fig. 2. 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.
A. Control Unit

The first IP, called AXI Neuron Control, is the control unit of the three remaining datapath IPs in Figure 2. In addition, it supports certain runtime adjustable settings like the batch size, the data format (ratio of integer and fractional bits), or the type of the activation function (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.

B. Datapath

The three datapath IPs support the conceptual mapping of an arbitrary batch size with up to n samples. Their implementation is visualized in Figure 3.

1) Batch Memory: Our Batch Memory allows the ARM cores to read and write arbitrary input and output data at any time. Together with the programmable BRAM controller of the IP, the control unit ensures that the correct input data is supplied to all parallel multiply-accumulate units (MACs). At the same time, the activations of the last section can be written into the memory. The crossbar between the memory hierarchy and the controller allows that each BRAM can play either the role of the input or the output, depending on the current processing state.

![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 · n neurons before a software intervention is required.](image)

2) DNN Coprocessor: The DNN Coprocessor is responsible for the transfer function which is the calculation of the weighted sum of inputs $z_1^{(j)}$. It can calculate 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 associated to one MAC unit in order to supply the weights. A FIFO stores up to one row (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.

Our MACs use a fixed point data format which is known as Q7.8 and stands for one sign bit, seven integer bits and eight fractional bits. This format is widely used and proven to be almost as accurate as float points for classification [10][12][21]. Note that the first pipeline stage of the MAC units does the multiplication using 16 bits, whereas the accumulation in the second pipeline stage is done with 32 bits. This ensures that the activation function input is provided with a full precision of 32 bits (e.g., Q15.16). It is further possible to adjust the ratio of the integer and fractional bits during runtime by shifting the decimal point before the activation function.

3) Activation function: Due to their fast implementation and precise outputs, some related works (e.g., [8]) favor precomputed activation function images that are stored in lookup-tables. However, such implementations of activation functions occupy a lot of valuable on-chip memory resources, either LUTRAM or BRAM, which could otherwise be used to store additional weights in order to increase the throughput. Furthermore, a dynamic change of the activation function at runtime would require a data intensive and slow transfer of new lookup-table contents. Therefore, we decided to implement the activation functions directly by using arithmetic operations. Modern activation functions like ReLUs are often times already piecewise-linear functions that only require a small number of cases and can thus be implemented efficiently in combinational logic. In our current design, we implemented the ReLUs and the sigmoid function in parallel. The latter implementation uses a piecewise linear approximation (PLAN) proposed by Amin et al. [22]. The desired function can be dynamically chosen by the AXI Neuron Unit. Furthermore, it is rather easy to extend the repertoire of activation functions by including additional function implementations. This allows the support of different activation functions for different neurons.

C. Throughput Optimization

Internally, all three datapath components of our design contain an extensive pipelining. Although these pipeline stages exist, Figure 3 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 $s_j \geq m$ and the coprocessor needs $s_j$ clock cycles for all MAC operations of the section, 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...
\[
\left[ \frac{s_{j+1}}{m} \right] \cdot s_j \cdot n + m \cdot c_a
\]

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
\[
t_{\text{calc}} \approx \frac{s_{j+1} \cdot s_j \cdot n}{m \cdot f_{pu}},
\]
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 \( j + 1 \) is
\[
t_{\text{mem}} = \frac{s_{j+1} \cdot s_j \cdot b_{\text{weight}}}{T_{\text{mem}}},
\]
given a size of \( b_{\text{weight}} \) for each weight and an actual memory throughput of \( T_{\text{mem}} \).

The optimal batch size \( n_{\text{opt}} \) that results in the maximum overall throughput is then achieved if \( t_{\text{calc}} \) is minimized by implementing as many processing units \((m)\) as possible and by setting \( t_{\text{mem}} = t_{\text{calc}} \), i.e. neither the memory interface nor the MAC units have to wait for data or requests. This optimal batch size \( n_{\text{opt}} \) can be calculated with
\[
n_{\text{opt}} = \frac{m \cdot f_{pu} \cdot b_{\text{weight}}}{T_{\text{mem}}},
\]

V. Experimental Results

In order to evaluate the batch processing concept and to experimentally determine the best performing batch size \( n \), we have tested our accelerator with different configurations against miscellaneous platforms.

A. SoC Implementation Platform

We chose the Zynq Evaluation and Development Board [23], short ZedBoard, for the implementation of our design. It represents a typical embedded SoC and features a XC7020 device with Artix-7 FPGA fabric. Table I depicts the resource utilization for different batch configurations. For the first three configurations (regular processing using just one sample, batch size 2, and batch size 4), we used the same bitstream and adjusted the batch size at runtime. Each design uses two clock domains: the memory interface (e.g., DMAs and FIFOs) is clocked with 133 MHz and the processing is done using a 100 MHz clock \( f_{pu} \).

Although our design scales nicely to all BRAMs on the device, we may not use all available DSP slices in every configuration. Since each MAC unit is connected to at least one BRAM, the number of DSP slices decreases as soon as the batch size rises.

<table>
<thead>
<tr>
<th>Resource (available)</th>
<th>LUT</th>
<th>LUTRAM</th>
<th>FF</th>
<th>BRAM</th>
<th>DSP48E1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Batch size 1</td>
<td>26518</td>
<td>1587</td>
<td>41353</td>
<td>140</td>
<td>114</td>
</tr>
<tr>
<td>Batch size 2</td>
<td>26105</td>
<td>1578</td>
<td>40129</td>
<td>100</td>
<td>106</td>
</tr>
<tr>
<td>Batch size 4</td>
<td>25251</td>
<td>1569</td>
<td>37627</td>
<td>100</td>
<td>90</td>
</tr>
<tr>
<td>Batch size 8</td>
<td>24602</td>
<td>1560</td>
<td>35125</td>
<td>100</td>
<td>80</td>
</tr>
<tr>
<td>Batch size 16</td>
<td>23953</td>
<td>1551</td>
<td>32623</td>
<td>100</td>
<td>70</td>
</tr>
<tr>
<td>Batch size 32</td>
<td>23324</td>
<td>1542</td>
<td>30121</td>
<td>100</td>
<td>60</td>
</tr>
</tbody>
</table>

B. Throughput Evaluation

For a fair comparison of both hardware and software, we have trained different neural networks architectures with multiple real-world data set. As many before us, we use the famous MNIST database of handwritten digits [24] 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 [25]. For this purpose, a person (who is wearing the smartphone) performed one of six activities (walking, walking upstairs, walking downstairs, sitting, standing, and lying). One sample of the data set is a 561-feature vector of time and frequency variables from different smartphone sensors (accelerometer, gyroscope, . . . ). A use case could be tracking of sport activities or, in a batch scenario, complete sequences of motions. The data set is divided into 7,352 training and 2,947 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 II. 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 I

<table>
<thead>
<tr>
<th>Resource (available)</th>
<th>LUT</th>
<th>LUTRAM</th>
<th>FF</th>
<th>BRAM</th>
<th>DSP48E1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Batch size 1</td>
<td>26518</td>
<td>1587</td>
<td>41353</td>
<td>140</td>
<td>114</td>
</tr>
<tr>
<td>Batch size 2</td>
<td>26105</td>
<td>1578</td>
<td>40129</td>
<td>100</td>
<td>106</td>
</tr>
<tr>
<td>Batch size 4</td>
<td>25251</td>
<td>1569</td>
<td>37627</td>
<td>100</td>
<td>90</td>
</tr>
<tr>
<td>Batch size 8</td>
<td>24602</td>
<td>1560</td>
<td>35125</td>
<td>100</td>
<td>80</td>
</tr>
<tr>
<td>Batch size 16</td>
<td>23953</td>
<td>1551</td>
<td>32623</td>
<td>100</td>
<td>70</td>
</tr>
<tr>
<td>Batch size 32</td>
<td>23324</td>
<td>1542</td>
<td>30121</td>
<td>100</td>
<td>60</td>
</tr>
</tbody>
</table>

### Table II

<table>
<thead>
<tr>
<th>Machine</th>
<th>ARM</th>
<th>Intel Core i7-5600U</th>
<th>Intel Core i7-4790</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Clock Freq. (MHz)</td>
<td>667</td>
<td>2600 - 3200</td>
<td>3600 - 4000</td>
</tr>
<tr>
<td>Cores (Threads)</td>
<td>2 (2)</td>
<td>2 (4)</td>
<td>4 (8)</td>
</tr>
<tr>
<td>L1 cache size (KB)</td>
<td>32</td>
<td>128</td>
<td>256</td>
</tr>
<tr>
<td>L2 cache size (KB)</td>
<td>512</td>
<td>512</td>
<td>1024</td>
</tr>
<tr>
<td>L3 cache size (KB)</td>
<td>—</td>
<td>4096</td>
<td>8192</td>
</tr>
<tr>
<td>Total RAM (MB)</td>
<td>512</td>
<td>8192</td>
<td>16384</td>
</tr>
<tr>
<td>Dual channel used</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>DDR3 controller peak bandwidth (GB/s)</td>
<td>4.2</td>
<td>12.8</td>
<td>25.6</td>
</tr>
</tbody>
</table>
Furthermore, all presented processors feature some variant of vector extension to accelerate floating-point intensive calculations through parallelism on instruction level. For the ARM Cortex-A9 this extension is called NEON [26] whereas both Intel CPUs can use SSE/AVX for this purpose [27]. In order to get the fastest runtime result on all presented platforms, the used framework of this work utilizes itself the BLAS [28] library for both execution and learning of DNNs. The library is individually configured and compiled for each of the used processors.

The throughput results for the DNN processing in software and hardware are depicted in Table III. In order to measure the execution 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 the ARM core) and read the cycle count before triggering the computation and after the computation is done. The execution times (i.e., time difference) are then given in milliseconds (ms) per samples. In addition, the shown results are also averaged over the complete test set of the used benchmark. Different neural network architectures were tested on all platforms. The architectures are taken or inspired from current research in the field. For example, the smaller network for MNIST was proposed in [29] while the second one is an extended version of that architecture with four additional hidden layers. Note that the software is using single-precision floating point numbers, whereas our hardware design uses the described Q7.8 fixed point format. Since most hardware accelerators utilizes the same format (cf. Section IV-B2), we reach similar accuracy results.

The best result for both our hardware and the software is 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 \( n_{\text{opt}} \) for the presented design is 12.66, assuming a constant number of \( m = 114 \) processing units clocked with \( f_{\text{pu}} = 100 \text{ MHz} \) and the used Q7.8 fixed point format. On the software side, we see the fastest runtime for the desktop machine with a utilization of 4 threads and dual channel (DC) memory. Even with optimized BLAS routines, 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

<table>
<thead>
<tr>
<th>Device</th>
<th>Configuration</th>
<th>4-layer network</th>
<th>8-layer network</th>
<th>4-layer network</th>
<th>6-layer network</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1,275,200 Parameters</td>
<td>3,835,200 Parameters</td>
<td>1,035,000 Parameters</td>
<td>4,351,050 Parameters</td>
</tr>
<tr>
<td>ARM Cortex-A9</td>
<td>non-BLAS</td>
<td>21.282</td>
<td>64.006</td>
<td>17.301</td>
<td>92.081</td>
</tr>
<tr>
<td>Intel Core i7-5600U</td>
<td>non-BLAS</td>
<td>1.334</td>
<td>4.116</td>
<td>1.184</td>
<td>5.869</td>
</tr>
<tr>
<td>Intel Core i7-4790</td>
<td>SC &amp; #Threads: 1</td>
<td>0.133</td>
<td>1.321</td>
<td>0.114</td>
<td>2.301</td>
</tr>
<tr>
<td>Intel Core i7-4790</td>
<td>SC &amp; #Threads: 4</td>
<td>0.071</td>
<td>1.108</td>
<td>0.053</td>
<td>3.015</td>
</tr>
<tr>
<td>Intel Core i7-4790</td>
<td>DC &amp; non-BLAS</td>
<td>1.005</td>
<td>3.167</td>
<td>0.877</td>
<td>4.515</td>
</tr>
<tr>
<td>Intel Core i7-4790</td>
<td>DC &amp; #Threads: 1</td>
<td>0.118</td>
<td>0.917</td>
<td>0.144</td>
<td>4.106</td>
</tr>
<tr>
<td>Intel Core i7-4790</td>
<td>DC &amp; #Threads: 4</td>
<td>0.057</td>
<td>0.569</td>
<td>0.045</td>
<td>1.205</td>
</tr>
<tr>
<td>Intel Core i7-4790</td>
<td>DC &amp; #Threads: 8</td>
<td>0.065</td>
<td>0.687</td>
<td>0.055</td>
<td>1.491</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MNIST a</th>
<th>HAR b</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network architectures: ( 784 \times 800 \times 800 \times 10 ) and ( 784 \times 800 \times 800 \times 800 \times 800 \times 800 \times 10 )</td>
<td></td>
</tr>
<tr>
<td>Network architectures: ( 561 \times 1200 \times 300 \times 6 ) and ( 561 \times 2000 \times 1500 \times 750 \times 300 \times 6 )</td>
<td></td>
</tr>
</tbody>
</table>

a Network architectures: \( 784 \times 800 \times 800 \times 10 \) and \( 784 \times 800 \times 800 \times 800 \times 800 \times 800 \times 10 \)
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 unless explicitly otherwise specified through non-BLAS. SC refers to a single channel memory configuration whereas DC utilizes dual channel.
configurations. 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 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 $2000 \times 1500$ 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). 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 the convolutional layers. Thus, they can achieve higher energy efficiency. For determining the energy consumption, we measured the system power for processing the 8-layer neural network, introduced in Section V-B, and the idle power for all platforms (see Table IV). 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 amper 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 IV.

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.

### D. Latency Evaluation

As mentioned earlier, the presented approach represents a trade-off between throughput and latency. Figure 4 compares the averaged latency of samples with the configured batch size.

![Fig. 4. Latency analysis for different batch sizes and network architectures. Latency is given in milliseconds and averaged over the test set of the network.](image)

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.

### E. Accuracy Evaluation

Since our accelerator utilizes the same Q7.8 fixed point data format as most related works [10][12][30][21], we obtain similar results concerning the accuracy. Concretely, when using ReLU as activation function we have a deviation of 10 mispredicted samples out of 10,000 (MNIST test set size) compared to the software prediction. This is exactly 0.1% or a one per thousand deviation to the software results that

### C. 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
were computed with single precision floating point numbers. However, since our design allows to dynamically adjust the number of integer and fractional bits at runtime, we were able to obtain the same accuracy as the software by switching to the Q4.11 format (one sign, four integer and eleven fractional bits). This confirms earlier results from [11, pp. 21] and shows that the fractional part is often times more important than the integer for the pure execution of DNNs.

For the sigmoid function, the accuracy evaluation is slightly different since we approximate the curve through piecewise-linear segments. Here, the Q7.8 format results in a deviation of 50 samples compared to the software which means a 0.5% misprediction rate. An negligibly better result was achieved with the Q2.13 format that led to 48 mispredicted samples, e.g. 0.48% misprediction rate.

VI. CONCLUSIONS

In this paper, we present a novel FPGA-based architecture for accelerating the processing of previously learned fully-connected deep neural networks. It is no longer feasible to map such deep networks with thousands of neurons and millions of weights directly onto FPGAs. Therefore, a time division multiplexing scheme is used to share processing units by loading the corresponding input values and weights from external memory. In order to reach a high processing throughput, the memory limitation of current embedded systems is circumvented by a so-called batch processing technique which shares different weights over multiple input samples. Together with a sophisticated pipelining, our approach outperforms other FPGA-based implementations for fully-connected layers by a factor of 11. Furthermore, once layer sizes reach typical deep network sizes, we outperform all software implementations on current processors while being at least 10 times more energy efficient. As far as the authors know, this is the fastest FPGA-based approach for processing fully-connected layers in deep neural networks. Despite all presented throughput optimizations, our approach remains fully flexible through a runtime adaption of the activation function and by an adjustable data format.

Future works on this topic might further increase the throughput of the proposed approach through a dedicated memory controller inside the FPGA fabric that achieves a comparable x86-based memory throughput.

REFERENCES