Analysis of Hardware Implementations to Accelerate Convolutional and Recurrent Neuronal Networks

Florian Kästner, Osvaldo Navarro Guzmán, Benedikt Janßen, Javier Hoffmann, Michael Hübner
Chair for Embedded Systems of Information Technology
Ruhr-University Bochum
Email: {Florian.Kaestner;Osvaldo.NavarroGuzman;Benedikt.Janssen;Javier.Hoffmann;Michael.Huebner}@rub.de

***Abstract*—Hardware platforms, like FPGAs and ASICs, turned out to be a viable alternative to GPUs for the implementation of deep learning algorithms, especially in applications with strict power and performance constraints. In terms of flexibility, FPGAs are more beneficial, while ASICs can provide a better energy efficiency and higher performance. Deep Learning is a subgroup of machine learning algorithms that has a major impact on modern technology. Among these algorithms, Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN) have been of great interest due to their accuracy in comparison to other methods. In this article, we conducted an analysis of the hardware implementation of these two popular network types. Different types of neural networks offer different opportunities to create an optimized hardware implementation due to their specific characteristics. Therefore, we split the analysis into two parts, discussing CNN and RNN implementations separately. Our contribution is an inside view on several hardware approaches and a comparison of their architectural characteristics. We aim to propose hints for their implementations.**

Keywords*— *FPGA; Recurrent; Convolutional; Neural Network; ASIC.

I. PREAMBLE

This article targets to collect solutions for hardware implementation of special types of Neural Networks (NNs) presented in [1]. The scope of this is, therefore, the discussion of hardware implementations of convolutional and recurrent neural networks. However, experience has shown that a short recapitulation of basics is beneficial. That is the reason why, this section is dedicated as a short review on the principles of NN.

NNs are a collection of connected units called $neurons$, which are typically organized in $layers$. In a NN each neuron is equipped with a linear and / or non-linear $activation function$, mapping the weighted inputs to the output. The layers can be connect in many different ways. For instance if they are connected in a sequential manner from the input to the output of the network, we are talking about a feedforward NN. The variation of the structure of neurons and their connections allow a wide field of NN types. The internal layers, i.e., not the one receiving the input or generating the output, are called hidden layers. If the network has several hidden layers, then it is said to be a Deep Neural Network. Both, Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) are implementations of these Deep Learning Networks (DLN).

II. INTRODUCTION

Over the last 30 years Deep Learning (DL) has become consistently more powerful providing accurate prediction or recognition. The breakthrough in the academic area was achieved in 2012 at the ImageNet Large Scale Visual Recognition Challenge [2] when a convolutional neural network (CNN), a specific type of Deep Neural Network (DNN), firstly won this challenge. After this breakthrough, the popularity and sets of application, not only of CNNs but of DL in general, has grown dramatically. In contrast to traditional machine learning principles the data representation models are trained without the need of handcrafted feature engineering. Within the structure and learning process of DNNs, features are extracted from the input data by itself rather than by humans. This is a mandatory advantage designing machine learned classifications or regressions for highly complex applications like detecting objects, for instance animals or everyday objects, from input images.

The resulting DNNs are capable of representing functions of high complexity with the help of connected simple, mostly hierarchical, components [3]. The supported complexity depends on the amount of layers and neurons or units inside each layer. Due to this structure, NNs can be trained to extract features at different levels of representations.

Two varieties of DNN have especially shown great potential to real life applications, CNNs and RNNs. CNNs are typically used for imagery classification [4], [5]. RNNs are suitable for time-variant problems such as speech recognition [6], due to their recursive structure. While the idea and structure of NNs are not new, their recent success is based on new training methods, namely backpropagation and pretraining. The increasing amount of collected data in combination with the processing power of modern compute platforms enables the exploitation of massive parallelism. Parallelism of data processing is a key principle regarding the training and also the inference phase, when only the feedforward path is active. The data flow driven architecture of DNNs allow coarse and fine grained parallelism. Additionally, distributed learning, batch processing and mini-batch processing increase the model's parallelism possibilities for accelerating the training of DNNs.

Nowadays Graphics Processing Units (GPUs) or clusters of GPUs are usually used to accelerate the training. Thus, highly complex models can be trained with a huge amount of data within days instead of weeks. GPUs are originally designed to accelerate data-flow driven graphic applications with massively parallel multi-core architectures. This architecture is also beneficial accelerating DL. Furthermore, improvements in the software infrastructure have lowered the effort of GPU implementations of DL algorithms. Frameworks like Caffe [7], Theano [8] or Tensorflow [9], in combination with efficient libraries like cuDNN[10] or cuBLAS [11] are designed to simplify the DNN implementation and enable a more efficient usage of GPUs.

However, GPUs have a high power consumption, compared to other processing units. Due to this disadvantage they are not suitable for an integration in most embedded devices, with strong power limitations. While service oriented applications, communicating via Internet and executing DNNs in the cloud, dont suffer due to that fact, applications with high safety criteria like autonomous driving need to have onboard processing of the inference phase of this DNNs to classify or predict. While GPU implementations have good framework-support and provide great performance capabilities, hardware acceleration implementations on Field Programmable Gate Arrays (FPGAs) or Application Specific Integrated Circuits (ASICs) represent an interesting and promising research area. However, currently such implementations are rarely used in industry.

In this paper we extend our previous work presented in [1] focusing on hardware implementations, their analysis and comparison. We further elaborate current trends in deep learning and how these trends could affect the popularity of hardware platforms like FPGAs. The rest of this paper is organized as follows. Section II-A describes the general structure and the improvements of popular and state-of-the-art CNNs. The same is done in Section II-B for RNNs. In Section III, we explain how the current trends could affect the choice of future platforms accelerating DNNs. The major part of this paper focuses on the analysis and comparison of hardware implementations of CNNs in Section IV and RNNs in Section V. The paper ends with a short conclusion in Section VI.

A. Convolutional Neural Networks

CNNs are a type of DNN for processing data that has a grid-like topology [3], [12]. CNNs are widely adopted for practical applications, especially in image processing tasks like object recognition or tracking. The input data of such tasks can be interpreted as a 2D grid of input pixels. The processing of these input pixels is depicted in Figure 1. One major advantage of CNNs is the absence of the need to flatten the input to a single dimension, avoiding information loss like positional relationships. CNNs usually consists of the following components: convolution layer, evaluation of a non-linear activation function, subsampling layer, fully-connected layer and classification or regression layer.

The convolution component extracts features from the input image with a set of adaptive filters called kernels. The convolution computation is done through a dot-product between the elements of the kernel and the input section of same size across the entire input frame and channels. Figure 2 shows a detailed demonstration of the computations applying a 2x2x2 convolution and max pooling on a 3x3x2 input image. Firstly, a multi-dimensional dot product is computed through the convolution layer consisting of 2 kernels. Each of these kernels possess dimension of 2x2x2. The depth of the input data, which can be defined as channels of an input image like Red-Green-Blue(RGB), has to be equal with the depth of the kernels. The resulting dimension of the output of the convolution layer is dependent on the dimension of the kernel and the used stride and padding behavior. Stride controls how the kernel convolves around the input data in a shifting manner. Padding is used to avoid a fast decrease of the output data adding data around the border of the input image. Thus, the height and width of the input image can be remained. The convolution layer in Figure 2 has a stride of one without padding. Therefore, the resulting output owns a dimension of 2x2x2, which can be described as a double-channel image or an output with two feature maps with a size of 2x2. Another possible method is to share one kernel for all channels to create decomposed images. The output of the dot-product is forwarded to an non-linear activation function, typically $sigmoid()$, $ReLU()$ or $tanh()$, which increases the nonlinear properties of the CNN. In practice, the rectified linear unit (ReLU), defined as $f(x) = max(0, x)$, is the most popular activation function used for CNNs due to its sparse activation, efficient computation and benefits regarding back propagation reducing the effect of vanishing gradients. Then, the pooling component, also called subsampling, reduces the spatial dimension of each feature map and keeps relative positional informations. The main purpose of this layer is generalization to avoid overfitting, by reducing the amount of parameters and keeping relative relationships. The most popular type of subsampling is max pooling as applied in Figure 2. In traditional CNNs like ImageNet [13], these three stages alternate several times. At the next stage of the feedforward path one or more Fully-Connected (FC) layers are applied. Typically, all neurons within these layers have full connections to the previous layer, while CNN layers are characterized through local connections and parameter sharing. However, both layers still compute dot products and therefore it is possible to replace fully-connected with convolution layer. At the last stage of a CNN a classification or regression layer is used to extract the desired information and build the quadratic cost function to train the network with the help of logistic or linear regression. The training of CNNs is done via a gradient-based backpropagation algorithm. Since the breakthrough in 2012, many researchers and companies modified the architecture and training methodologies for CNNs improving generalization and precision. The developers of

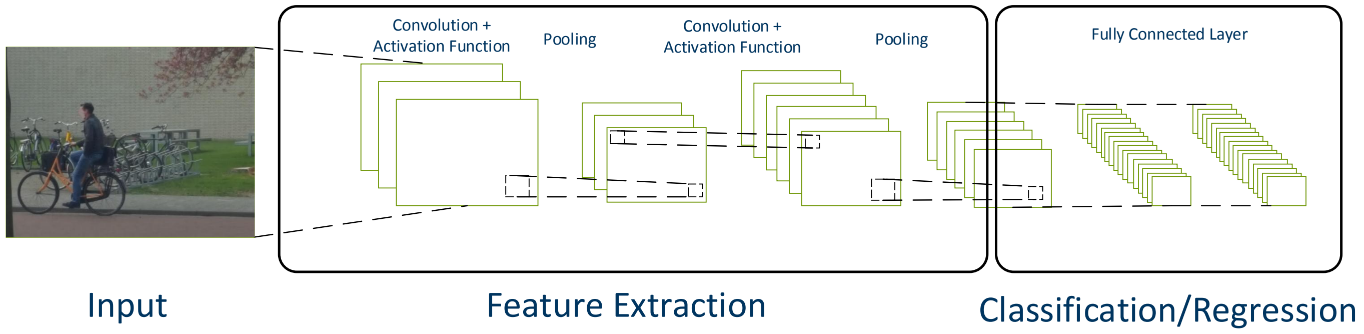

Fig. 1: Common processing flow of a convolutional neural network [3].

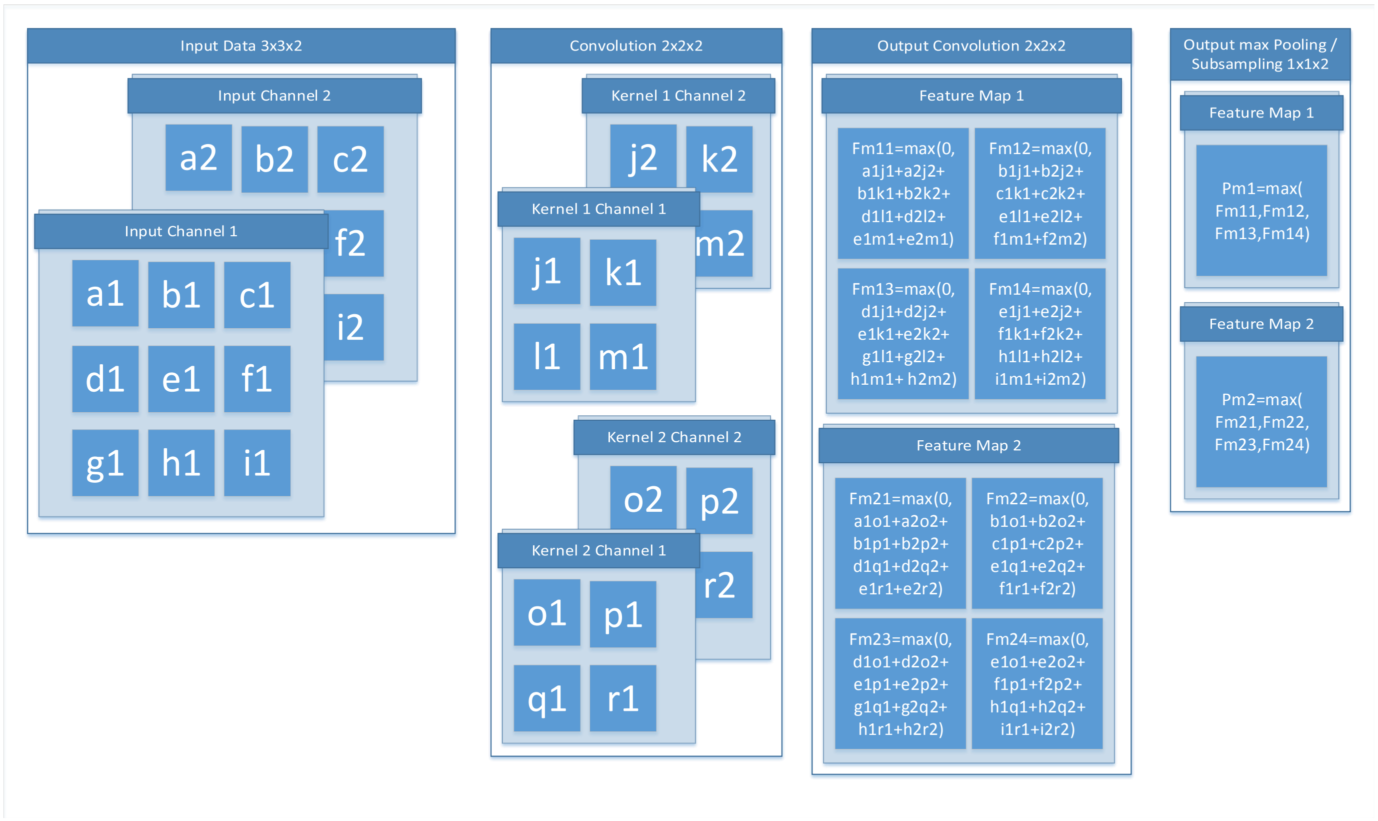
Input Data 3x3x2
Input Channel 2
a2 b2 c2
f2
i2
Input Channel 1
a1 b1 c1
d1 e1 f1
g1 h1 i1
Convolution 2x2x2
Kernel 1 Channel 2
j2 k2
m2
Kernel 1 Channel 1
j1 k1
l1 m1
Kernel 2 Channel 2
o2 p2
r2
Kernel 2 Channel 1
o1 p1
q1 r1
Output Convolution 2x2x2
Feature Map 1
Fm11=max(0, a1j1+a2j2+ b1k1+b2k2+ d1l1+d2l2+ e1m1+e2m1)
Fm12=max(0, b1j1+b2j2+ c1k1+c2k2+ e1l1+e2l2+ f1m1+f2m2)
Fm13=max(0, d1j1+d2j2+ e1k1+e2k2+ g1l1+g2l2+ h1m1+ h2m2)
Fm14=max(0, e1j1+e2j2+ f1k1+f2k2+ h1l1+h2l2+ i1m1+i2m2)
Feature Map 2
Fm21=max(0, a1o1+a2o2+ b1p1+b2p2+ d1q1+d2q2+ e1r1+e2r2)
Fm22=max(0, b1o1+b2o2+ c1p1+c2p2+ e1q1+e2q2+ f1r1+f2r2)
Fm23=max(0, d1o1+d2o2+ e1p1+e2p2+ g1q1+g2q2+ h1r1+h2r2)
Fm24=max(0, e1o1+e2o2+ f1p1+f2p2+ h1q1+h2q2+ i1r1+i2r2)
Output max Pooling / Subsampling 1x1x2
Feature Map 1
Pm1=max(Fm11,Fm12, Fm13,Fm14)
Feature Map 2
Pm2=max(Fm21,Fm22, Fm23,Fm24)

Fig. 2: Demonstration of the computations by applying a 2x2x2 convolution with max pooling on a 3x3x2 input image

VGGNet [14] (2014) proved that keeping CNNs simple and deep is a useful method in order to improve their performance to a certain degree. Karen Simonyan and Andrew Zisserman archieved the lowest error rate with 16 layer CNN using 3x3 kernels with a stride of one and 2x2 max pooling with a stride of two in every layer [14]. Another approach to improve the precision of CNNs was presented by Szegredy et al. [15] in the same year with a similar error rate compared to VGG-16, but with a model size which consists of only 6 million instead of 140 million parameters. The introduction of inception modules leads to the evolution of GoogleNet. The basic idea of GoogleNet is to break with the traditional sequential order of layers. The inception module proposed in [15] consists of a 3x3 max pooling, a 3x3 convolution and a 5x5 convolution executed in parallel. The output of these components are concatenated in the depth dimension. In order to reduce dimensions to 1x1 convolutions to each component are applied. Furthermore, instead of FC layers, average pooling layers are used. Thus, the amount of parameters, mainly arised from the FC layers, has been significantly decreased. As a result the decreasing size of the model and the increasing data parallelization possibility is beneficial for accelerating training and inference phase. However, the current state of the art CNN for object classification, detection and localization is ResNet [16]. These CNN introduced by He et al. 2015 is by far the biggest network of the mainly used ones with a total size of 152 layer and 60 million parameters. Due to the huge size of the network even applying ReLU activation units the vanishing gradient problem is exacerbated. Therefore, the basic idea of ResNet is to sequentially apply shortcuts over convolutional layer. The output of the convolution is then added to the original input. Thus, a slightly different representation of the input data is created. The resulting residual mapping simplifies the backpropagation optimization and allow models to be even deeper.

B. Recurrent Neural Networks

Recurrent neural networks (RNNs) can be utilized for recognition, production or prediction problems [17]. Unlike conventional NNs, they can include temporal information in the processing, as they can handle information relation between sequential data for sequences of arbitrary length. RNNs extend the concept of conventional acyclic feedforward NNs by maintaining a hidden state. This ability is enabled by cycles inside the network, and it is a key feature for tasks such as speech and handwriting recognition [18]. The cycles inside the network are connections from the output of a layer looping back, either to itself, or to a previous layer. The feedback connection enables the network to represent previous information as activation [19]. Thereby, the activation of the hidden state depends on previous activations [20]. In [21], Schmidhuber describes RNNs as more powerful general computers in comparison to acyclic feedforward NN. According to Schmidhuber, RNNs can learn from arbitrary data inputs, as well as create such data as output. Even a mixture of sequential and parallel data can be processed. Similar to other DNNs, RNNs can benefit from a massive parallelism in processing, as offered by todays computing platforms. In comparison to Markov chains, which can also handle dependencies in time between sequential input data, the state space of RNNs is growing slower for growing context windows. In a Markov model the growth of state space is exponential, for RNNs, it is quadratic at most [22].

One of the first approaches for NN to include temporal information, is the work of Hopfield [23]. He describes an early NN that allows temporal dependencies of the neurons state and an encoding of temporal information of data. One of the first approaches in the direction of todays RNNs was done by Jain et al. They developed a partially RNN to learn character strings [24]. According to [24] the architecture of RNN can be anything between a fully interconnected network and a partially interconnected network. In fully interconnected networks, every neuron is connected to every other neuron, as well as to itself via feedback connections. This structure is depicted in Figure 3. However, in contrast to Figure 3, for a fully interconnected RNN, there are no distinct input and output layers.

In the 90s, there have been difficulties to train RNNs for applications whose data includes dependencies over long temporal intervals. The results from Bengio et al. in [25], indicated that it is more likely for the learning process to take into account dependencies with short temporal character, rather than those with long-term dependencies. In [17], Bengio et al. discuss this issue for a better understanding of the

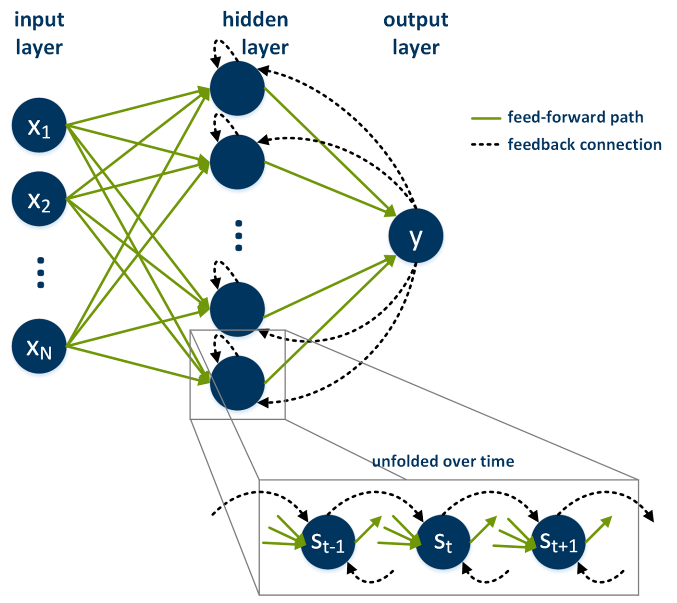

Fig. 3: General architecture of RNNs with optional input and output layer.

problem. They suggest alternative optimization algorithms to gradient-descent, which showed encouraging results. Within their paper, they reveal the vanishing gradient problem and the exploding gradient problem. Pascanu et al. investigated this problem further and proposed a regularization term that hinders the error signal to vanish [26].

The first contribution to a network to overcome the issue of learning long-term dependencies, was made by Hochreiter and Schmidhuber in 1997 [19]. They proposed a new recurrent network architecture, together with a gradient-based learning algorithm, called long short-term memory networks (LSTM). These networks can be categorized as a sub-group of RNNs. The network architecture contains an input layer, a hidden layer, and an output layer. The hidden layer contains so called memory cells and is fully connected. A LSTM cell architecture is depicted in Figure 4.

Another notable approach was presented by Schuster and Paliwal [27]. They proposed bidirectional recurrent neural networks (BRNNs). BRNNs are designed to overcome the issue of choosing the right time for the output delay to include future input data. Therefore, Schuster and Paliwal proposed to split the neuron state to cover positive time direction and the negative time direction. This means that the network structures includes positive and negative delays. Their results show a significant improvement for the classification of phonemes from the TIMIT speech database over other types of RNNs and multi-layer perceptron networks.

More recently, Cho et al. proposed a new type of hidden units, called gated recurrent unit (GRU) [28]. The approach was motivated by the LSTM approach, but instead of three or four gating units in LSTM, depending on the implementation, it uses only two gating units. In [20], Chung et al. show that GRUs can gain advantage over LSTM-based RNNs.

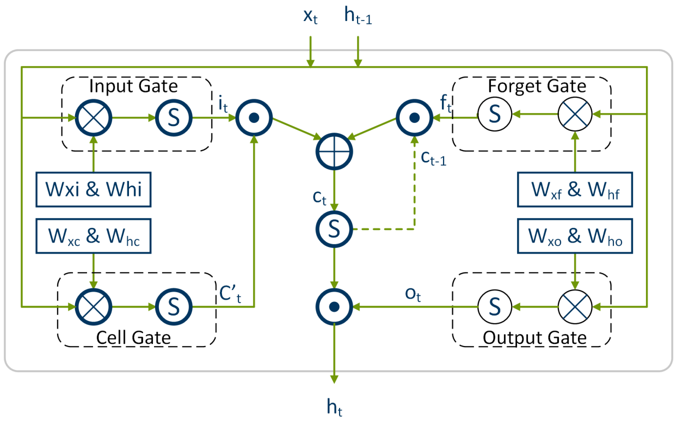

Fig. 4: LSTM architecture with four gated units. The S denotes the Sigmoid function [18].

III. Hardware Acceleration on GPU and FPGA

The advantage in energy efficiency of FPGAs in comparison to other computing platforms is well known [29], [30], [31]. However, in the area of DNNs, GPUs are a more prominent platform for implementations in comparison to FPGAs. This dominance is, among other aspects, based on a higher computing performance and more efficient programmability in e.g., CUDA programming language. The advantage in performance is based on the heavy use of floating-point operations of current DNN implementations. In [32], Nurvitadhi et al. compare the implementation of RNNs on FPGAs, CPUs, GPUs and ASICS. Without the usage of batch processing, the FPGA implementation performs better than its counterpart on CPUs and GPUs. This is due to its customizable hardware architecture. With the usage of batch processing, the implementation on CPUs and GPUs achieve a better performance, however, their utilization is not as optimal, leading to a worse efficiency.

In [33], the authors present their follow-up work. They analyzed the capabilities of upcoming FPGA technology, and the latest developments of DNN, to identify benefits of FPGA-based implementations. For their evaluation, they compared implementations on Intels Arria 10 and Stratix 10 FPGAs to implementations on Nvidias Titan X Pascal GPU. According to the authors, the latest developments in the field of DNNs are reported to be an increase in efficiency, compact data types, and network sparsity. Architectures with more efficient data paths are necessary, as the size of current DNNs is growing, due to a coherence in inference accuracy. Moreover, the data processing can be optimized by exploiting zero values in the network weights and biases, called sparsity. It is necessary to take advantage of zero values by avoiding their computation [33]. Recent publications [33] have shown a significant improvement in representing the data inside DNNs with less bits, while still maintaining a high accuracy.

Modern GPUs are often based on a single-instruction multiple-thread (SIMT) compute model, meaning that multiple threads are processed in parallel, executing the same instructions on different data. This is the case for the compute model of OpenCL, where a single program counter is used for a group of threads, called wavefront [34]. Thus, thread-level divergence is an issue, as only one execution path can be followed at a time. Therefore, a homogeneous processing is beneficial for GPU implementations. Moreover, GPUs usually support a fixed set of native data types. Thus, compact data types with only a few or single bits are less likely beneficial for GPU implementations and can even become a performance disadvantage. Modern FPGAs on the other hand offer an increasing amount of on-chip memory. For instance, Intels Stratix 10 offers up to 28 MB [35], and Xilinx Virtex UltraScale+ offers more than 45 MB [36]. Moreover, their maximum frequency is increasing due to improvements in the production process, as well as new features, such as Intels HyperFlex. In addition, the number of hardwired DSPs cores, equipped with native floating-point support, is increasing. Furthermore, the achievable bandwidth is increasing too, for instance the usage of high-bandwidth memory (HBM) in future FPGA generations.

Based on these properties of GPUs and FPGAs, as well as the trends of current DNN development, the authors of [33] foresee a possible performance benefit for DNN implementations on FPGAs, in addition to the performance/Watt advantage.

Nurvitadhi et al. results indicate that for single-precision floating-point implementations of DNNs, FPGAs still fall behind the performance of GPUs. However, as stated earlier, the results show an advantage for FPGAs when looking at performance/Watt. To evaluate sparsity, the Nurvitadhi et al. evaluated AlexNet with matrices with 85 % sparsity. This means that only 15 % of the elements are non-zero. Due to the shortcomings of GPUs thread-divergence, FPGAs can offer a benefit. In fact, the results show that the Stratix 10 FPGA outperforms the Titan X GPU in matrix multiplication for DNNs with pruning for implementations with a clock frequency of 500 MHz and more. The evaluation of an extreme case of compact data types, binary neural networks with data types of 1bit width, reveal the disadvantage of a fixed set of natively supported data types. The results of Nurvitadhi et al. show that the Stratix 10 FPGA outperforms the Titan X GPU in matrix multiplication for binarized DNNs for all implementations. This is even the case for the implementation on the lower-cost Arria 10.

Another DNN type with compact data types are ternary DNNs, which use a ternary representation of the network weights, e.g., +1, 0, -1. For their evaluation, Nurvitadhi et al. analyzed Ternary ResNet. Their implementation takes advantage not only of compact data types but also sparsity in layers. On average, they report 51 % sparsity of the weights, and 60 % sparsity of the neurons. In their experiments, the FPGA implementation, running at 450 MHz, was able to beat the implementation on the Titan X GPU in terms of performance, as well as performance/Watt.

As indicated by the results of Nurvitadhi et al., in addition to the advantage in performance/Watt, there is a good chance for next-generation FPGA technology to beat GPUs for DNN implementations.

Therefore, within this paper we will analyze implementation details of the two prominent kinds of DNNs, namely CNNs and RNNs, and discuss their properties and benefits.

IV. Analysis of CNN Implementation

As mentioned in Section II-A, CNNs are data flow oriented and offer great possibilities to design hardware accelerators and even coprocessors exploiting fine and coarse grained parallelism. Filtered Connections (FC) and convolution layers can be seen as big matrix multiplications. Within the convolution layer of the example shown in Figure 2, 64 Multiplications and Accumulate Computations (MACs) can be executed in parallel. The amount of operations without data dependencies which are able to be executed in parallel increases, due to the kernel and input data size. However, several challenges arise designing an efficient hardware accelerator for CNNs. Obviously, modern CNNs with a huge amount of layers like ResNet are not suitable to implement it as a whole. Therefore, methods and architectures have to been found facing a trade-off between resource utilization and performance. Furthermore, the biggest challenge focuses on the memory allocation and access, which is the main bottleneck in every architecture.

While image processing cores have already been extensively studied and widely used on FPGAs, to the best of our knowledge Sankaradas et al. firstly designed and implemented a coprocessor able to execute and accelerate the inference phase of an entire CNN [37]. The authors of [37] bypass the bottleneck of memory access and allocation with the help of off-chip memory with a large bandwith. The off-chip memory, including 4 banks of DDR2 SDRAM of 256MB, is connected to a Xilinx Virtex-5 FPGA via PCI. The basic unit of this architecture is a handcrafted 2D convolver using Single Instruction Multiple Data (SIMD) processing elements called Vector Processing Elements (VPEs). These VPEs consist of fixed amount of chained digital signal processing processor (DPS) units optimized for an efficient execution of MAC operations and a First In First Out (FIFO) buffer as can be seen in Figure 5. Furthermore, these VPEs are organized in clusters whereby the output of each VPE tile is added to the previous one with the help of an additional DSP unit. The advantage of dynamic reconfiguration is not used due to the fact that the FPGA is used as a prototyping platform. Thus, the size of the clusters and amount of DSP units inside each VPE is fixed. Although the VPEs are programmable to skip convolutions, the size of the baseline kernel has to be previously determined. Thus, bigger kernels are not suitable for executing convolution on this architecture. The clusters also include subsampling and non-linear activation function primitives performed by look-up tables. The input data is streamed in a sequential manner and no intermediate data is stored on the on-chip memory. To further increase the communication performance, Sankaradas et al reduce the data precision from floating point to fixed point operations of 20 bits. Later studies [38] proved that reducing the data precision in a certain region does not impact the accuracy loss significantly. Sankaradas et al. achieved with the first coprocessor, capable to accelerate a complete CNN on hardware, a performance of $3,37GMAC$ per second with a total power consumption of $11W$. This baseline coprocessor is a very well designed architecture. However, this architecture still does not use the parallelization possibilities properly and seems to be very inefficient applied to a CNN with varying kernel and sampling sizes. On the other side, this static architecture is perfectly suitable applied to simple and homogenous CNNs like VGG.

In contrast to the architecture described in [37] the authors of [39] abandon the option of using external memory and a persistent connection to a local host computer in their hardware accelerator design focusing on mobile embedded applications. Jin et al. use a Xilinx Zynq-7000 device including a dual ARM Cortex-A9 core and an Artix-7 FPGA to implement their architecture consisting of 2 main components, namely a memory router and a collection network. A collection is defined as a group of operator blocks performing arithmetic operations like convolution. Within a single pass through the collection, an output of a 1-to-1 convolution plane is applied. Each collection can be executed in parallel with an interval of 1, which means that every clock cycle, an output datum is produced. To control the data flow between each operator, each collection owns a router. This router is also able to bypass an incoming data stream from neighbored collections. The operators inside each collection are arranged in a sequential manner similar to the VPE described above. The other major component of this architecture is the memory router, which consist of 3 AXI Direct Memory Access (DMA) IP cores. These DMAs, performing the memory access via AXI4-bus coupled with AXI4 Stream, acting as a gateway to the collection. Furthermore, the memory router is able to distribute incoming streams to not occupied collections. Unfortunately, Jin et al. did not describe properly how the streams are built or how the exact operator blocks are designed in order to synchronize kernels and input data. The authors mention that the peak performance of 40Gops/s consuming less than 4W significantly decreases, applying deep CNNs producing deep feature maps according to the huge amount of intermediate data. However, the smart routing technique and reduced number of global connections is a promising method to execute inference phase of CNNs keeping intermediate data in a pipelined stream.

Although the above described architectures are carefully designed and programmable in a certain degree, they are very static and inflexible to introduce changes, which leads to inefficiency implementing different types of CNNs. Redesigning the hardware to improve different stages of different CNNs produces high engineering effort. Therefore, newer approaches and design flows implementing custom CNNs on hardware try to overcome this issue with the help of High Level Synthesis (HLS). HLS is an automated design process producing hardware description at the Register-Transfer-Level (RTL) from a higher level of abstraction, namely the algorithm-level. Thus, Verilog or VHDL code can be automatically generated from

C/C++, SystemC or OpenCL code. Typically, the architectural design of the resulting hardware architecture can be influenced with a set of constraints and directives. This method ensures a fast design space exploration and an easy customization of the hardware to accelerate DNNs. The authors of [40] use Vivado HLS from Xilinx to implement a five-layer CNN on a FPGA including 2 convolutions with 5x5 kernels and 2 average pooling layer. Instead of data streaming combined with smart routing or external memory, Zhou et al. use mainly the on-chip memory of a Virtex7 to store parameters, intermediate and input data. After noticing that the onboard Block RAM (BRAM) has only two ports preventing convolution to be executed in parallel sufficiently, Zhou et al. switch to a sea of registers through a line buffer chain. In combination unrolling the most inner loop of the convolution they were able to execute 25 11bit-MAC operations in parallel. The hardware accelerator is driven with a clock frequency of 150 MHz. Therefore, this design has a theoretical maximum peak performance of $3.75 GMACs$. However, this paper shows the advantage of HLS designing custom CNN accelerators due to the easy handling and the hardware generation speedup with state-of-the-art HLS tools.

The authors of [41] use the properties of HLS to effectively perform a Design Space Exploration optimizing FPGA hardware accelerators for CNNs. This research is based on the roofline performance model described in [42] relating the system performance to off-chip memory traffic and the peak performance. The goal of the design space exploration is to achieve a certain computation to communication ratio in order to reach the computational roof of the attainable performance. Due to that goal the authors of [41] first started to optimize the computation of the convolution by generating a series of valid CNN implementations. Thus, specific directives provided by Vivado HLS, namely unrolling and pipelining, are used to generate a series of valid hardware accelerators of CNNs with an equivalent functionality with the help of loop scheduling and loop tile size enumeration. More precisely Zhang et al. investigated in the impact of tile size selections in order to find the computational roof of a convolution with the given loop scheduling. To reduce the amount of memory access, local memory promotion is proposed for keeping the intermediate data on-chip as long as possible. In order to increase the reusability of intermediate data, a polyhedral-based optimization framework is used, identifying all legal loop transformations. As a result Zhang et al. performed a Design Space Exploration of a convolution layer with multiple input channels and feature maps to compute a set of Pareto points. Thus, a hardware implementation can be chosen achieving the best performance in combination with a suitable communication to computation ratio. The synthesized IP-core is coupled to two ping-pong buffers for each input and output AXI4 bus. The off-chip data transfer is managed by two data transfer engines to perform DMA access. In order to gain the needed bandwidth, the amount of AXI4 bus interfaces has been adjusted. However, the multi-layer CNN accelerator suffers due to a chosen global unroll factor. This fact leads to inefficient hardware utilization in heterogeneous CNNs especially by performing the feedforward path of FC layers. The 5 layer CNN hardware accelerator is implemented on a Xilinx Virtex 7 device achieving $61.62 GFLOP/s$. The hardware is working with $100 MHz$ consuming $18.61 W$. Instead of fixed point Zhang et al. only use floating point operations in their design.

To further increase the performance of FPGA based accelerators of CNNs, Li et al. investigated the opportunities to customize every type of layer due to their specific characteristics. Therefore, the authors of [43] try to optimize the memory access method and the level of parallelism depending on the layer type and their properties. While the system architecture and the parallelism space exploration is very similar to the architecture presented in [41], the major difference is the fact that Li et al. treat convolution and FC layers differently in terms of optimizing the level of parallelism and off-chip memory access. The FC layers are handled as big matrix multiplications and divided into small scale matrix multiplications. Furthermore, computing the optimal data parallelism Li et al. took the limitation of the hardware resources, namely DSP and BRAM units, directly into account. Due to the fact that FC layers are very memory-bounded and less computation-bounded caused by their dense interconnections, a batch-based processing method for FC layers has been proposed. The basic idea of this approach is to continuously run the FC feedforward path without waiting for data. Therefore, Li et al. compute a batch size matching the computing pattern for the FC layers in order to increase the operations executed in parallel without increasing the needed bandwidth. Hence, the 8-layer AlexNet hardware implementation on a Virtex 7 device is working concurrently in a pipeline structure achieving $565.94 GOP/s$ with an energy consumption of $30.2 W$.

While Li et al. used floating point operations, the authors of [44] used a data quantification strategy reducing the bit-width down to 8-bit without increasing the accuracy loss significantly. The data quantification strategy aims to search for radix point positions for each layer for a given bit-width while the CNN is still trained with floating point operations on a GPU. By producing a histogram of the logarithmic values of the feature maps, initial radix point positions are set. Then, a greedy strategy is used to optimize the positions layer by layer. After the best accuracy is achieved with a set of radix point positions, the CNN was fine-tuned converting back to floating point format. However, Guo et al. used a 24 bit-width to store intermediate data avoiding an increasing accuracy loss. Another major benefit of the architecture described in [44] is the fact, that this design is reconfigurable during runtime. The architecture supports kernel sizes of 1x1, 3x3 and 5x5. To configure the accelerator Guo et al. extend their previous work [45] with flexible set of instructions. Furthermore, a compiler is provided for mapping the network descriptor to the instructions. This compiler performs a set of optimizations to achieve a good computation communication ratio with the use of all available hardware resources. The coprocessor was tested with different CNNs such as GoogleNet, VGG-16 or

SquezzeNet achieving $137GOP/s$ on a Xilinx Zynq-7020 device with a power consumption of $3.5W$.

As a novel idea for improving the performance of the system, [46] presents the *Global Summation*, designed to minimize resource consumption on the latest layers in favor of the overall performance. In contrast, [47] presents several techniques to improve the control method of the process, among several improvements, for instance, it is important to mention the input reuse and the concatenation of data. The Input reuse, consists on extending the information to convolution modules even though if the amount of available ports is smaller than that of free convolution modules. Those convolution modules that have access to the port can share the information to those who are idle. The concatenation of data profits of the Q8.8 format. The Q8.8 format uses 16 bits in a 32 bitstream, so that 2 different data words can be transmitted in a unique stream.

Another approach is proposed in [48]. Yuan et al. design a coarse-grain array to accelerate the arithmetic operations. The basic unit of the hardware accelerator is the Neural Element (NE) including 9 multiplication and 1 adding operation units. This basic unit is optimized to convolve 3x3 kernels. Figure 6 shows the coarse grain array formed by 16 NEs, which are arranged to form a Configuration Neural Block (CNB). To reconfigure this array two configurable routing units are used, namely Filter conncetions (FCs) and Input Connections (ICs). As their name suggest, these switching ressources are responsible to route kernel weights and input data to their desired location. Furthermore, 5 adding operation units are placed adding the output of multiple different NEs. In this way other kernel sizes can be supported from 1x1 to 12x12. Additionally, the overall architecture owns 2 CNBs to further increase the parallelism. Like the authors of [45] Yuan et al. provide a compiler taking trade-offs due to the computation to communication ratio and on-chip resources into account to optimize the routing. However, this architecture does not fully utilize the on-chip resources applied to other kernel sizes than 3x3 like 5x5 due to its coarse grained nature. Although the non-used processing elements can be turned off during execution and the smart arrangement and routing are producing a low degree of overhead, the circuit of the coarse grain array cannot be reconfigured as this architecture is designed as an ASIC implementation under a TSMC 65nm CMOS process. The ASIC is clocked by 100MHz and achieve a peak performance of $57,6GOP/s$ with a power consumption of $107mW$.

ASIC-based approaches offer great performance and low power capabilities; however, they lack flexibility and have a larger design flow process. The approaches based on this platform are highly customized for an application. Under this category, [49] introduced $Eyeriss$, accelerator for deep CNNs that aims for energy efficiency. The accelerator consists on an SRAM buffer that stores input image data, filter weights and partial sums to allow fast reuse of loaded data. This data is streamed to a spatial array of 14x12 processing engines (PE), which compute inner products between the image and filter weights. Each PE consists on a pipeline of three stages that computes the inner product of the input image and filter weights of a single row of the filter. The PE also contains local scratch pads that allow temporal reuse of the input image and filter weights to save energy. There is also another scratch pad in charge of storing partial sums generated for different images, channels or filters, also with the aim of reusing data and saving energy. To optimize data movement, the authors also propose a set of input Network on Chips (NoC), for filter, image and partial sum values, where a single buffer read is used by multiple PEs. This approach was implemented in a $65nm$ CMOS, achieving a frame rate $44.8fps$ and a core frequency of $250MHz$.

The work in [50] proposes an analog-digital hybrid architecture for CNNs targeting image recognition applications, focusing on high performance at low power consumption. The architecture consists on a pulse-width modulation circuit to compute the most common operations of a CNN and a digital memory to store intermediate results. Additionally, the design makes use of a time-sharing technique to execute the operations required by all the connections of the network with the restricted number of processing circuits available in the chip. This architecture was implemented with a $0.35\mu m$ CMOS pieces. The paper reports an execution time of $5ms$ for a network with 81 neurons and 1620 synapses, an operation performance of $2GOPS$ and a power consumption of $20mW$ for the PWM neuron circuits and $190mW$ for the digital circuit block.

The work in [51] proposes Yodann, an ASIC architecture for ultra-low power binary-weight CNN acceleration. In this work, a binary-weight CNN is chosen for implementation because limiting a CNN's weights to only two values (+1/-1) avoids the need of expensive operations such as multiplications, which can be replaced by simpler complement operations and multiplexers, thus reducing weight storage requirements. This also has the advantage of reducing I/O operations. Moreover, this approach implements also a latch-based standard cell memory (SCM) architecture with clock-gating, which provide better voltage scalability and energy efficiency than SRAMs, at the cost of a higher area consumption. The architecture was implemented using UMC $65nm$ standard cells using a voltage range of $0.6V - 1.2V$. The article reports a maximum frequency of $480MHz$ at $1.2V$ and $27.5MHz$ at $0.6V$ and an area of 1.3 MGE (Million Gate Equivalent).

Recently, in February 2017, the company ST Microelectronics published a System-on-Chip (SoC) design in FD-SOI 28nm accelerating CNNs on embedded devices. Desoli et al. [52] achieved a theoretical peak performance of $676GOP/s$ with a peak efficiency of $2.9TOPS/W$ (Tera Operations Per Second/ Watt). The interesting components of this architecture are the hardware convolutional accelerators supporting kernel kompression and the on-chip reconfigurable data-transfer fabric. The data-transfer is managed by 10 fully configurable DMAs. Additionally, a configurable stream switch guides 53 input and 40 ouitput streams to their desired locations like one the eight Convolution Accelerators (CAs). Hence, the CAs can be arbitrarily chained. Therefore, a configurable accelerator framework is used in order to configure the SoC during design-time. The CA is equipped with 4 stream interfaces, namely feature stream input, kernel stream input, intermediate data stream input and an output stream. The feature line buffer can store up to 12 channels with up to 512 pixels and provides 36 read ports fetching data in parallel. Moreover, up to 484 kernels values can be stored inside the kernel buffer. Convolution is done by 36x16x16bit overlapping and column-based fixed-point MACs able to convolve 4 kernels in parallel followed by an adder tree. This design is able to consistently reuse available hardware ressources due to its fully scalable configuration. It is worth to mention that Desoli et al. complete the SoC with a set of extensions like pheripherals, high-speed interfaces for imaging and a chip-to-chip multilink in order to offer a basic platform for sophisticated image processing and to connect several devices to a bigger accelerator.

In the same year, the Envision architecture was presented by Moons et al. The authors of [53] focus on increasing the energy efficiency of their CNN hardware accelerator with the help of subword-parallel Dynamic-Voltage-Accuracy-Frequency Scaling (DVAFS) enabling Envision to achieve 10TOPS/W in 28nm FDSOI. Therefore, Moons et al. extend the principle of DVAS with reusing inactive arithmetic cells at reduced precision. As a result, frequency and voltage of the processor can be significantly decreased compared to DVAS.

Although ASIC implementations offer greater performance capabilities in combination with a lower energy consumption they lack in terms of flexibility. While homogenous networks are great for ASIC implementations, executing heterogeneous ones owning different kernel sizes and feature maps often lead to an inefficient use of the hardware or parallelism possiblities. Thus, the application of FPGAs can overcome this problem. While the above described architectures aim to be realized in future as hard wired coprocessors, the utilization of the FPGA serves as a rapid prototyping platform. The development of new variants of CNNs is rapid and is improved due to their specific applications. Thus, the size and structure of the network also can be adjusted to specific applications. As mentioned earlier in this section, HLS is a promising way to accelerate the design time of hardware accelerators. Additionally, newer HLS tools like Vivado HLS provide a methodology to perform design space exploration with the help of tcl-scripts automatically. This offers great possibilities to create frameworks automatically generating bitstreams executable on reconfigurable devices. Solazzo et al. propose a web-based framework in [54] interfacing directly with Vivado HLS. Although HLS synthesis is not complicated, the synthesis time can be huge. Hence, the main contribution of [54] is to predict the resource utilization of a given network configuration. Unfortunately, no optimization methodologies are used. The C/C++ language was not designed taking parallelism into account, though. Hence, OpenCL is more native way to describe parallelism. Xilinx also provide a HLS tool called SDAccel focusing on the synthesis of OpenCL code. DiCecco et al. use this tool in [55] to add FPGA support to the Caffe deep learning framework. While OpenCL support in Caffe is already given DiCecco et al. designed their OpenCL implementations to be suitable to many architectures rather than only GPUs. While optimizations are CNN specific and can be easily performed by the SDAccel tool DiCecco et al. focusing on the integration and synchronization of FPGA accelerators in Caffe. Hence, Wang et al. describe in their work [56] methods to improve CNN accelerators with the help of SDAccel and OpenCL optimizing the pipelining structure and level of parallelism. All the approaches described in this section aim to maximize the performance of CNNs by explot-ing the level of parallelism and bandwidth for computing the feedforward path of CNNS. However, the resulting architectures heavily depend on the available target hardware ressouces and the desired application CNN. The less ressources are available, the harder is the task of keeping intermediate data on the on-chip memory avoiding bottlenecks. Furthermore, the paralellism level is dependent on the amount of feature maps, which have to be evaluated. The roofline-model and the optimization exploration, which was described regarding CNN in [41], is a good baseline for evaluating custom CNN accelerators. In combination with data quantification techniques, discussed in [44], sophisticated routing methods [39] and embedding methodologies into common deep learning frameworks [56], hardware implementations, esspecially on FPGAs, of CNNs could become a promosing alternative to GPU CNN acclerators not only applied on embedded devices.

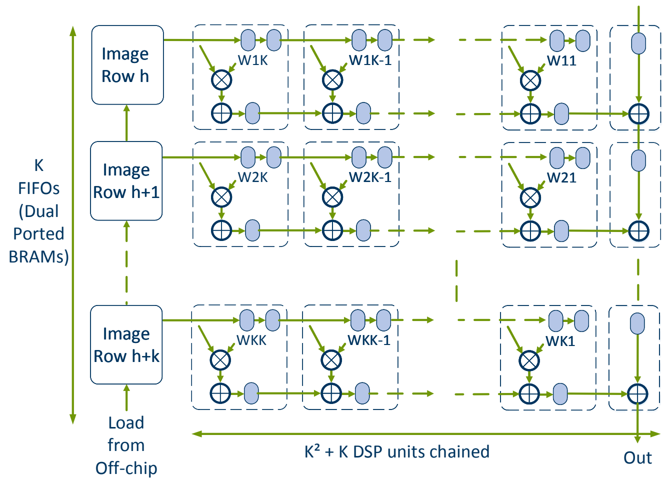

Fig. 5: A VPE array implementing the primitive 2D convolver unit by Sankaradas et al. [37].

V. Analysis of RNN Implementations on FPGAs

In general, similar to the optimizations regarding CNN implementations, two possibilities exist to increase the overall performance of RNNs regarding the inference phase on FPGAs. However, due to the recursive structure of RNNs and the differences of the computations inside the neurons, the optimization has to be handled differently. On the one hand the computations, which must be done while passing the input through the network can be optimized. This includes adjusting

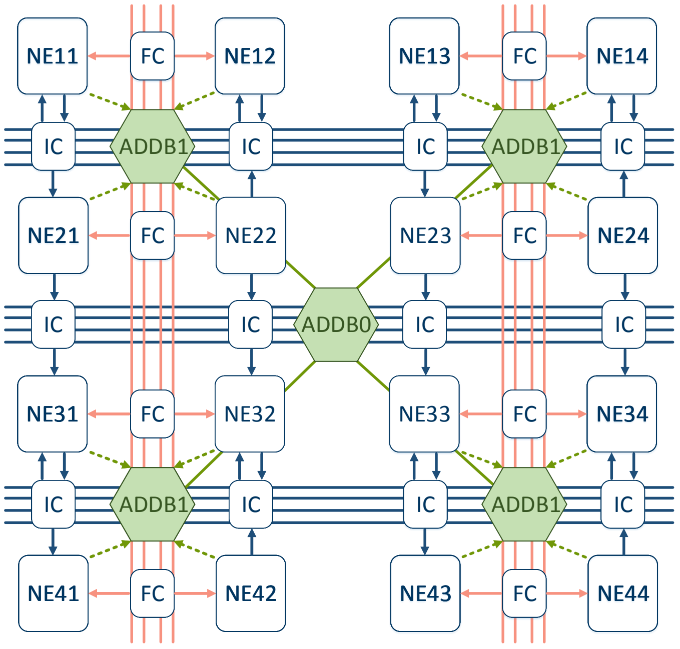

Fig. 6: Coarse grain array design to acclereate CNNs by Yuan et al. [48].

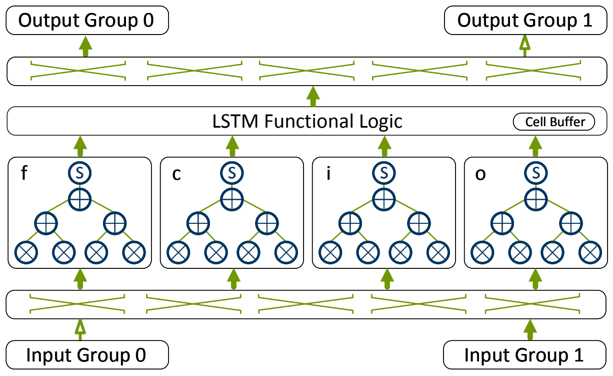

Fig. 7: Architecture of the accelerator for a LSTM-RNN proposed in [18].

the level of parallelism due to the hardware constraints, the choice of hardware accelerator module and approximations of computation, in case of an acceptable minimal loss of precision. On the other hand, the data communication, which includes weight parameters and inputs, between the FPGA and an external memory, like DRAM, can be optimized to improve the throughput. This is necessary because typically the FPGA on-chip-memory, BRAM or distributed memory, is too small to store all parameters of real-life RNNs. Thus, the developer of FPGA accelerators for RNNs have to consider both optimization paths. Moreover, to increase the overall performance both optimization paths must collaborate to avoid bottlenecks.

In [18], the authors focused on both optimization paths previously mentioned. Regarding the computation optimization, the authors profiled a typical LSTM-RNN inference structure and found out that the main bottleneck is caused by the computations, which mainly include floating point multiplication and addition, inside each LSTM gate. To optimize these operations, they split the computations in each gate into tiles, each of them carry out a portion of the inference process. Then, the execution among tiles is pipelined to optimize the throughput while the inner loops within each tile are unrolled and executed in parallel to improve the latency. The second most executed block of operations are the activation functions. These were replaced with a piecewise linear approximation of nonlinear function (PLAN), which was originally introduced by Amin et al. [57]. This approach consist of simple additions and shifting operations, which can be more efficiently implemented in hardware. This comes at a price of a small loss of accuracy, which can be ignored.

Regarding the optimization of communication requirements, the authors tackle the issue of irregular data access, which is caused by the transposition and tiling of the computation optimization described previously. This irregularity in the memory accesses causes a significant overhead on the communication between the DRAM and the FPGA. To solve this issue, the matrices used in the computations are modified offline in such a way that they can be accessed sequentially during the inference phase. Furthermore, 2 input buffers and 2 output sets of buffers working in a ping-pong fashion were added to further improve communication. Finally, a data dispatcher was added to maximize the the use of the bandwidth between the DRAM and the buffers.

To implement these optimization methods, they separated the computation scheme into four LSTM gate modules: the input, forget, cell and output. These models are shown in Figure 7. These modules receive tiled input vectors from the input buffers in parallel through a crossbar and carry out the inference process. The multiplications within this process are carried out in parallel within each LSTM gate module. Then, the results are summed up using an addition tree. the summed up results are delivered next to a *LSTM Functional Logic* module, which executes the remaining operations, such as element-wise multiplication, addition of gate vectors, activation, etc. Moreover, the current state of the LSTM cell is stored in a module called *Cell Buffer*. Finally, the complete results are sent to the output buffers through another crossbar. The Accelerator is designed with Vivado HLS and the system is implemented on a XILINX VC707 board with a Virtex7 FPGA chip. A DDR3 DRAM is used as external memory, which holds the parameters of the LSTM-RNN, as well as the inputs and outputs. A MicroBlaze processor is used to control the accelerator and to measure execution time. An AXI4 and an AXI4Lite are used for communication between the modules and to transfer commands, respectively. The evaluation shows that the system achieves a maximum performance of 7.26 GFLOP/s,

A different architecture is proposed by Chang et al. [58]. Like in the previously mentioned architecture, they focus on optimizing the operations of the LSTM inference process and the communication. The architecture of their approach is shown in Figure 8. This architecture has 3 LSTM gates, each of which carry out either an hyperbolic tangent or a logistic sigmoid function, and an element-wise multiplication module. Another similarity with the previous approach is the simplification of these functions, using piecewise linear approximation. This means, the functions where segmented into linear functions, i.e., $y = ax + b$, which are easier to be implemented on hardware. The values of each of these functions are stored offline in the Configuration Registers module. Each of these linear functions where implemented using a MAC operation and a comparator. In contrast to the previous design, Chang et al. [58] used fixed point 16-bit operations for MAC operations resulting in 32 bit values for further operations. While fixed-point computations are far more efficient for hardware implementations, a loss of accuracy has to be considered, which was denoted as a maximum of 7.1%. Furthermore, the gating computations are separated into two sequential steps. The input and cell gate computations are done in parallel as well as the output and forget gate computations. As a result, the output of the LSTM module is provided after 3 sequential steps. Thus, the coarse grained parallelism is not fully exploited. For communication optimization purposes, the authors of [58] use a combination of memory mapping and streaming interface. Therefore, four direct memory access (DMA) cores are used to access the external DRAM and reshape the data to be forwarded through 8 AXI4-Stream modules, which activity depends on the current routing, and are buffered with FIFOs. In comparison to the communication architecture presented by Guan et al. [18], the methodology is equal but the differences in implementation details are mainly arise because of the different coarse grained parallelism of the LSTM accelerator. The design is implemented on a Zedboard with Zynq 7020 FPGA from Xilinx. The architecture was tested with a character level language model, which, given a character from a text as input, it predicts the next character. The network consisted of 2 LSTM cells, each of which including 128 hidden units, and running with a frequency of $142 MHz$. The performance was outlined to 264.4 million operations per second.

Lee et al. [30] used two LSTM-RNNs to build a speech recognition system. The general structure of the system is shown in Figure 9. The system uses a LSTM-RNN for acoustic modeling and another one for character-level modeling. The acoustic modeling LSTM-RNN analyzes the input speech and calculates the probability of occurrence of each character. The character-level modeling LSTM-RNN calculates the probability of the occurrence of each character given a previous one. Similarly, another language model calculates the probabilities at the word level. Finally, these results are combined using the N-best search algorithm.

Figure 10 shows the hardware architecture that implements this system. The architecture has a LSTM cell and an output tile, which are used intermittently for the acoustic modeling and for the character-level modeling operations, according to a control signal. The output of the LSTM cell is stored in a context memory, and is used in the next block of operations and by the N-best search algorithm.

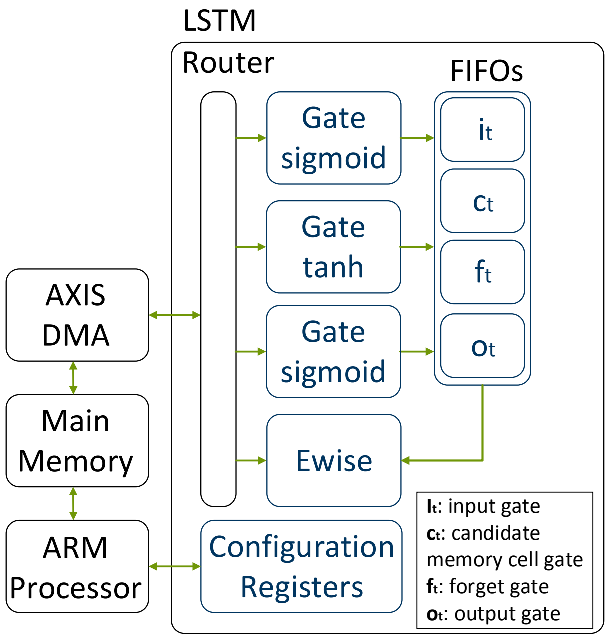

Fig. 8: Block diagram of the accelerator for a LSTM-RNN proposed in [58].

In contrast to [58] and [18], the authors of [30] use a retrained based method to reduce the word-length of the weights. As a result, they achieve a quantification to 6 bits per weight. Due to this fact, for the desired speech recognition application all weight parameters can be stored on the on-chip-memory (BRAM) of the XC7Z045 FGPA device from Xilinx without loss of precision differences of retraining and hardware implemented inference computation. Without the requirement to optimize the communication from an external DRAM the focus of this work is the accelerator module optimization. Although the proposed LSTM accelerator module is separated into two different modules, the parallelism and task granularity differs from those proposed in [18] and [58]. All matrix-vector multiplications are computed inside one (PE) array, which consist of 512 PEs. The remaining computations including evaluating activation functions are done using an additional block called Extra Processing Unit (EPU). The outputs of the PE array are buffered and forwarded to the LSTM EPU. As mentioned above, the design benefits from a high level of fine-grained parallelism. However, the architecture is not comparable to the other design as no communication bottleneck has to be handled and the authors focus on the optimization of the whole speech recognition algorithm while considering real-time constraints for the desired application.

Besides, Ferreira et al. [59] follow a modular extensible architecture, they assume a similar reduction of communication complexity as assumed in [30]. A diagram of the architecture is shown in Figure 11. In contrast to [30] the word-length of the weights are 18 bits, which leads to a high

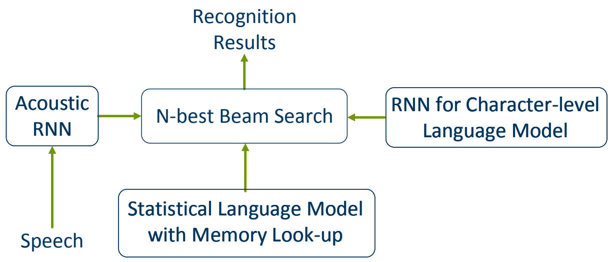
Recognition Results
Acoustic RNN
N-best Beam Search
RNN for Character-level Language Model
Statistical Language Model with Memory Look-up
Speech

Fig. 9: Speech recognition system proposed in [30].

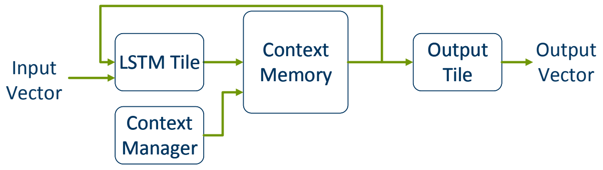

Fig. 10: Hardware architecture for the speech recognition system shown in Figure 8, as proposed in [30].

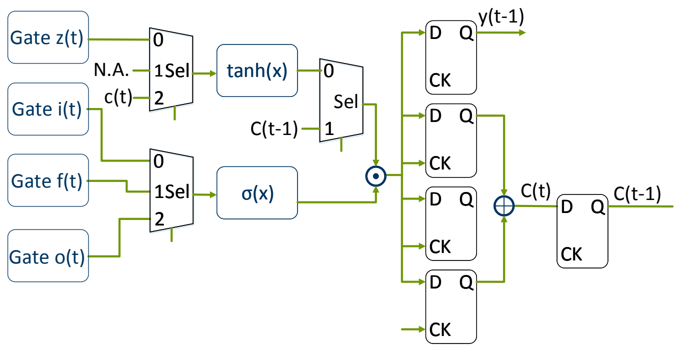

Fig. 11: Hardware architecture LSTM-RNN as proposed by Ferreira et al. [59].

utilization of the DSP48E1 slices of the FPGA, and they are stored in LUTRAM. On the one hand, this distributed memory represents the fasted way of accessing the weights due to the physical closeness to the accelerator and the unlimited simultaneous port access to each LUTRAM array. In contrast BRAM supports a maximum of two port access. On the other hand, this method consumes a high amount of resources especially when the implemented network consists of many layers and LSTM-tiles. The architecture does not explore full parallelism in a coarse-grained manner. The matrix-vector multiplications are done in parallel for all four gates. Instead of directly implementing the activation functions $tanh()$ and $sigmoid()$, polynomial approximations where carried out, in order to find equivalent polynomials within an acceptable error range. The strategy Least Maximum Approximation was used to find the optimal polynomial for each activation funtion. Due to the negligible small amount of additional clock cycles needed for elementwise multiplication and polynomial approximations of $tanh()$ and $sigmoid()$, each of these computations where carried out in one single hardware module. The gate outputs are forwarded to a multiplexer and further routed to the desired hardware module. The design was implemented on a Xilinx XC 7Z020SoC device with different amounts of neurons per layer. The network size is not allowed to extend 31 neurons per layer due to the limited number of DSP-slices on the device. The frequency of the hardware accelerator was adjusted to the maximum with respect to the layer size. Thus, the design is capable achieving 4534.8 MOP/s, which is 17 times more than the performance reached in [58].

Han et al. [60] proposed a hardware accelerator for a speech recognition system based on LSTM-RNNs, which involves a compression method that significantly decreases the LSTM-RNN's size while keeping an acceptable accuracy. The compression method consists on pruning and quantization. The pruning is carried out by removing the weights from the LSTM-RNN that do not contribute to the prediction accuracy. Moreover, the linear quantization strategy was used to generate weights represented by 12bit integers instead of 32 bit floating point values. Then, the compressed LSTM-RNN is implemented in hardware. A general diagram of the architecture is shown in Figure 12. Unlike the work presented in [30], where all weight parameters can be stored in on-chip memory due to a 6-bit quantization, the 12-bit quantization of this work made it necessary to store this data in external memory. Two 4GB DDR3 DRAMs where used for this purpose. Similar to [18], input and output buffers where used in a ping pong manner such that the communication and the computation are overlapped. Furthermore, this approach differentiates from the ones previously described by implementing modules, called channels, each of which can process a voice vector independently. Each channel consists of several Process Elements (PE), which carry out the inference process of the LSTM-RNN. Besides the usual challenges that implementing a LSTM-RNN on hardware implies, the pruning method used in the compression process introduces the problem of dealing with sparse matrices. To deal with this issue, a PE called Activation Vector Queue (ActQueue) is used to balance the workload among the rest of the PEs. The ActQueue PE consists of several FIFOs, which store elements from the input voice vector, and delivers the input elements to the PEs across all the channels, such that fast PEs are not blocked by slower ones. Furthermore, similarly to the approaches already described, linear approximations where used to implement the activation functions, and the matrix multiplication operations where carried out in parallel within a PE. The system was evaluated with a XILINX XCKU060 FPGA running at a frequency of $200MHz$ and compared against a software approach. The experimental results showed a throughput of $282GOP/s$

Li et al. [61] addressed the problematic of developing a proper model to approximate or evaluate the probabilities of finding out the next word in a sentence. They affirm that RNN is one of the best suited approaches to deal with this topic, on behalf of statistical methods such as n-gram finding or decision trees. They mentioned that, although RNN are more complex, need higher computation capabilities and long training times,

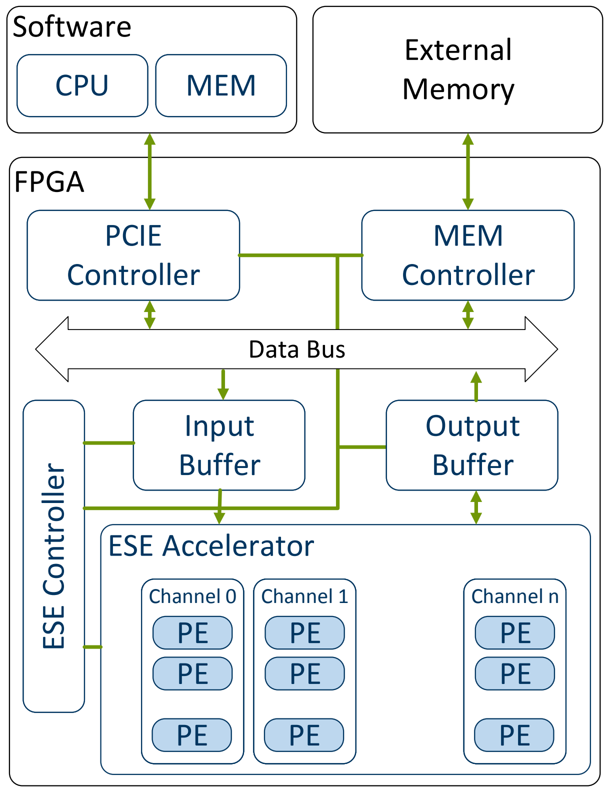

Fig. 12: Hardware accelerator for speech recognition system, with multiple channels for multiple voice vectors [60].

the results are more reliable since they can consider previous states and thus create a dependency model.

These computational shortcomings can be handled with the use of hardware approaches and hence they develop a model for FPGA acceleration. They delegate the starting and initialization of the weights and the acceleration system to the CPU of the host system. For the memory, they use an external memory, with 16 Dual Inline Memory (DIM) Modules and 1024 banks. These characteristics are profitable since the bank conflicts turn to be low. They also implement an Application Engine Hub (AEH) to manage the instructions from the host.

The PE are associated in major sets the computational Engines (CE). These CEs are separated into two classes with the only difference in the activation function. The first class is for the hidden layers, all PE in the hidden layers belong to this first class. The second class is dedicated to the output layer, and here the PEs are grouped in one of the many CE. The authors affirm that these characteristics are suited for expanding the number of PEs, depending on the requirements.

Previous approaches require the utilization of an on board-Block RAM, which the authors of [61] try to avoid in their approach. Therefore, they came up with the idea to utilize a hardware supported multithreading architecture, generating a thread for a defined matrix operation. The PEs are in charge of processing dedicated operations of a major task, that is, each task is separated in threads that are processed by the PEs.

As mentioned before, the CE is the manager of the memory accesses and can fetch information as needed or in a burst mode, in addition, this model has the advantage of data reuse by storing results either on a weight register or in a bias register. This is useful for the stage where the PEs are ready to start with the activation function processing.

Using these hardware configuration, Li et al. [61] affirm they obtain an improvement on the parallelism between layers presented on a previous paper and also the computation efficiency, in this matter, a fixed-point data conversion is used, under the premise that NN handle and correct the imprecisions of a truncation by themselves.

The authors also compared the performance with a multi-core CPU and a GPU. The CPU had 12 cores at a frequency of 2.3GHz, while the GPU 512 cores at 772MHz. The frequency for the FPGA was 150MHz. As expected, the GPU turned out to be the one with the lesser execution time, while the CPU the one that required more time. The energy analysis showed another perspective, where the FPGA was the one consuming less energy and the GPU turned second with a consumption level as high as almost 7 times the FPGA.

Renteria-Cedano et al. [62] investigate on an special type of NN: the Nonlinear Auto-Regressive Exogenous networks (NARX), which they state that they have been shown to converge much faster than other implementations of RNN. They implement the NARX network over an FPGA using only one hidden level, thus having a total of three layers, with the goal of linearizing microwave power amplifiers.

The authors present the hardware implementation where the PEs are able to carry out the operations necessary: multiply, add, accumulate and activation function. The flow starts when the information is provided to the PE, this process is according to its internal parameters, the multiplication afterwards is done in parallel. Finally, after the addition of the bias, the selected activation function, the hyperbolic tangent on a Taylor series implementation, takes place for those PE in the hidden layer. It is also important to mention that the authors used floating point for every operation and the general management of the NN is done according to Mealy type finite state Machine.

Another variation of a RNN is the Hopfield Network. Atencia et al. [63] recognize the benefits of implementing this type of network on FPGAs, since, as they mention, concurrency is an own characteristic of these networks and at the same time, they have a reduced number of neurons, which suits the capabilities of the state of the art for FPGAs at the time they developed their research. In this approach, a neuron consist of a RAM linked directly to a MAC, from this point the data is feed forwarded to a substractor, a multiplier, an accumulator and finally to the activation function before it is outputted and processed by a multiplexer. The weights for each neuron are calculated in an external software and stored in the neurons memory, these weights are then supplied as operands to a MAC together with a status value provided from the multiplexer. The next step is a bias substraction, its multiplication by the discretized value of the ordinary differential equation characteristic of Hopfield networks and the addition to a previous state value, finally the activation

TABLE I: COMPARISON OF IMPLEMENTATION APPROACHES

Approach	Device	Power [W]	Performance	Type
Sankaradas et al.[37]	Virtex-5	11	3.37 GMAC/s	CNN
Jin et al.[39]	Zynq-7000	4	40 GOP/s	CNN
Zhou et al.[40]	Virtex-7	n/a	3.75 GMAC/s	CNN
Zhang et al.[41]	Virtex-7	18.61	61.62 GFLOPS	CNN
Li et al.[46]	Virtex-7	30.2	565.94 GOP/s	CNN
Guo et al.[44]	Zynq 7020	3.5	n/a	CNN
Guan et al.[18]	VC707	20	7.26 GFLOP/s	RNN
Chang et al.[58]	Zynq 7020	2	0.264 GOP/s	RNN
Ferreira et al.[59]	Zynq 7020	2	4.538 GOP/s	RNN
Han et al.[60]	Kintex	41	282 GOP/s	RNN
	UltraScale	41	282 GOP/s	RNN
Li et al.[61]	Virtex-6	25	9. 6GOP/s	RNN

function is enforced with a LUT representing the tanh() function.

All the approaches described in this section aimed to maximize the performance of RNNs by parallelizing key operations of the inference phase of the network. Although the matrix operations carried out by RNNs are highly parallelizable with custom hardware architectures, there are still issues, which are still not solved entirely. The main issues we found were the representation of the RNN's parameters and the memory bottleneck, which are interrelated. For instance, [30] avoided the use of external memory by using 6 bit integers to represent the LSTM-RNN's parameters. While in that case the loss of accuracy brought by this quantization was deemed acceptable, this is highly dependent on the application and input data. Furthermore, whenever external memory was used, a special scheme had to be developed, such as ping-pong buffers, in order to minimize the bottleneck.

VI. CONCLUSION

As outlined in the previous sections, there are many possibilities to accelerate CNNs or RNNs with the help of hardware modules. Each of these hardware implementations have a significant speedup in comparison to a CPU implementation. From our perspective, there are several reasons making a fair comparison of all hardware architectures not possible. One of the reasons for that is, that the size of the networks mostly depends on the application as well as the real-time constraints. The analysis in Section IV and Section V focuses on benefits or deficits regarding the acceleration performance derived from architectural similarities and differences rather than comparing numbers. Table I summarizes the measurable dimensions of the CNN and RNN implementations on different FPGA architectures.

However, in our opinion, presenting only a comparison of quality metrics such as performance or throughput would not show a complete view of the advantages and drawbacks of each approach. For instance, the performance of almost all described hardware accelerators can be improved with simple approaches, like increasing the number of processing elements and bandwidth. Thus, an important influence factor of the FPGA prototypical architectures is the underlying platform. However, the FPGA technology and HLS support is improving rapidly. While the FPGA prototypical architectures and ASIC designs aim to low energy consumption on embedded devices reusing static processing blocks, future work should exploit the hardware reconfiguration capabilities of FPGAs in order to further increase flexibility needed regarding different layer types and sizes. It is also worth to mention that the combination of RNNs and CNNs, called RCNNs [64] (not to be confused with R-CNNs, which is Region based CNNs for object localization), is a promising approach improving the accuracy of object and scene detection. For example, the platform proposed in 2017 by Shin et al. [65] combines a CNN and a RNN in a single configurable processor to harness the advantages from both networks: the image recognition capabilities from the CNNs and the ability to recognize sequential dependencies between the data from the RNNs. This combination, however, brings new challenges, since their hardware requirements can be very different. For instance, the convolutional layers of CNNs require a large number of operations over a relatively small number of weights, while LSTM cells require a smaller number of operations over a much larger number of parameters.

Finally, in the industrial environment, is of relevance to mention that Google's Cloud Tensor Processing Unit (TPU)[66] is a viable alternative for accelerating NN implementations to those presented in this article. These TPUs are ASICs designed especially for machine learning applications, and according to Google, have been used successfully on different projects, e.g., Alpha Go or Street View.

ACKNOWLEDGMENT

This work was done under support of CONACyT (Grant 359472).

REFERENCES

[1] J. Hoffmann, O. Navarro, F. Kästner, B. Janssen, and M. Hübner, "A survey on cnn and rnn implementations," in *PESARO 2017, The Seventh International Conference on Performance, Safety and Robustness in Complex Systems and Applications*, 2017.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105.

[3] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[4] W. Zhao, S. Du, and W. J. Emery, "Object-based convolutional neural network for high-resolution imagery classification," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. PP, no. 99, pp. 1–11, 2017.

[5] H. J. Jeong, M. J. Lee, and Y. G. Ha, "Integrated learning system for object recognition from images based on convolutional neural network," in *2016 International Conference on Computational Science and Computational Intelligence (CSCI)*, Dec 2016, pp. 824–828.

[6] T. He and J. Droppo, "Exploiting lstm structure in deep neural networks for speech recognition," in *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, March 2016, pp. 5445–5449.

[7] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 2014, pp. 675–678.

[8] Theano Development Team, "Theano: A Python framework for fast computation of mathematical expressions," *arXiv e-prints*, vol. abs/1605.02688, May 2016.

[9] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org.

[10] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[11] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, and E. S. Quintana-Orti, "Evaluation and tuning of the level 3 cublas for graphics processors," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, April 2008, pp. 1–8.

[12] Y. Sugomori, *Java Deep Learning Essentials*. Packt Publishing Ltd., 2016.

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," Web site: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf [Last accessed: 24 September 2017], pp. 1097–1105, 2012.

[14] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014. [Online]. Available: http://arxiv.org/abs/1409.1556

[15] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," Web site: http://dblp.uni-trier.de/rec/bib/journals/corr/SzegedyLJSRAEVR14[Last accessed: 21 September 2017], 2014.

[16] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," http://arxiv.org/abs/1512.03385 [Last accessed:7 June 2017], 2015.

[17] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 157–166, Mar 1994.

[18] Y. Guan, Z. Yuan, G. Sun, and J. Cong, "Fpga-based accelerator for long short-term memory recurrent neural networks," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2017, pp. 629–634.

[19] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, Nov 1997.

[20] J. Chung, Ç. Gülçehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *CoRR*, vol. abs/1412.3555, 2014.

[21] J. Schmidhuber, "Deep learning in neural networks: An overview," *CoRR*, vol. abs/1404.7828, 2014.

[22] Z. C. Lipton, "A critical review of recurrent neural networks for sequence learning," Web site: http://arxiv.org/abs/1506.00019 [Last accessed: 12 September 2017], 2015.

[23] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proceedings of the National Academy of Sciences*, vol. 79, no. 8, pp. 2554–2558, 1982, accessed: 09 Jun 2017. [Online]. Available: http://www.pnas.org/content/79/8/2554.abstract

[24] L. C. Jain and L. R. Medsker, *Recurrent Neural Networks: Design and Applications*. CRC Press, Inc., 2001.

[25] Y. Bengio, R. D. Mori, G. Flammia, and R. Kompe, "Global optimization of a neural network-hidden markov model hybrid," *IEEE Transactions on Neural Networks*, vol. 3, no. 2, pp. 252–259, Mar 1992.

[26] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training Recurrent Neural Networks," http://adsabs.harvard.edu/abs/2012arXiv1211.5063P [Last accessed: 21 September 2017], Nov. 2012.

[27] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, Nov 1997.

[28] K. Cho, B. van Merrienboer, Ç. Gülçehre, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," Web site: http://arxiv.org/abs/1406.1078 [Last accessed: 13 September 2017], 2014.

[29] A. K. Jain, D. L. Maskell, and S. A. Fahmy, "Are coarse-grained overlays ready for general purpose application acceleration on fpgas?" in *2016 IEEE 14th Intl Conf on Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*, Aug 2016, pp. 586–593.

[30] M. Lee, K. Hwang, J. Park, S. Choi, S. Shin, and W. Sung, "Fpga-based low-power speech recognition with recurrent neural networks," in *2016 IEEE International Workshop on Signal Processing Systems (SiPS)*, Oct 2016, pp. 230–235.

[31] M. Al Kadi, B. Janssen, and M. Huebner, "Fgpu: An simt-architecture for fpgas," Web site: http://doi.acm.org/10.1145/2847263.2847273 [Last accessed: 20 September 2017], New York, NY, USA, pp. 254–263, 2016.

[32] E. Nurvitadhi, J. Sim, D. Sheffield, A. Mishra, S. Krishnan, and D. Marr, "Accelerating recurrent neural networks in analytics servers: Comparison of fpga, cpu, gpu, and asic," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2016, pp. 1–4.

[33] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra, and G. Boudoukh, "Can fpgas beat gpus in accelerating next-generation deep neural networks?" in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: ACM, 2017, pp. 5–14.

[34] (2017, Sep.) Opencl 1.2 specification. version: 1.2. document revision: 19. Web site: http://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf [Last accessed: 16 September 2017].

[35] (2017, Sep.) Stratix 10 gx/sx device overview. version: S10-overview 2016.10.31. Web site: https://www.altera.com/en_US/pdfs/literature/hb/stratix-10/s10-overview.pdf [Last accessed: 15 September 2017].

[36] (2017, Sep.) Ultrascale architecture and product data sheet: Overview. v2.11. Web site: http://arxiv.org/abs/1506.00019 [Last accessed: 12 September 2017]. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf

[37] M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H. P. Graf, "A massively parallel coprocessor for convolutional neural networks," in *2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, July 2009, pp. 53–60.

[38] R. Doshi, K. W. Hung, L. Liang, and K. H. Chiu, "Deep learning neural networks optimization using hardware cost penalty," in *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2016, pp. 1954–1957.

[39] J. Jin, V. Gokhale, A. Dundar, B. Krishnamurthy, B. Martini, and E. Culurciello, "An efficient implementation of deep convolutional neural networks on a mobile coprocessor," in *2014 IEEE 57th International Midwest Symposium on Circuits and Systems (MWSCAS)*, Aug 2014, pp. 133–136.

[40] Y. Zhou and J. Jiang, "An fpga-based accelerator implementation for deep convolutional neural networks," in *2015 4th International Conference on Computer Science and Network Technology (ICCSNT)*, vol. 01, Dec 2015, pp. 829–832.

[41] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15. New York, NY, USA: ACM, 2015, pp. 161–170.

[42] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009.

[43] H. Li, X. Fan, L. Jiao, W. Cao, X. Zhou, and L. Wang, "A high performance fpga-based accelerator for large-scale convolutional neural networks," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2016, pp. 1–9.

[44] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, and H. Yang, "Angel-eye: A complete design flow for mapping cnn onto embedded fpga," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. PP, no. 99, pp. 1–1, 2017.

[45] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang, "Going deeper with embedded fpga

platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '16. New York, NY, USA: ACM, 2016, pp. 26–35.

[46] N. Li, S. Takaki, Y. Tomiokay, and H. Kitazawa, "A multistage dataflow implementation of a deep convolutional neural network based on fpga for high-speed object recognition," in *2016 IEEE Southwest Symposium on Image Analysis and Interpretation (SSIAI)*, March 2016, pp. 165–168.

[47] A. Dundar, J. Jin, B. Martini, and E. Culurciello, "Embedded streaming deep neural networks accelerator with applications," *IEEE Transactions on Neural Networks and Learning Systems*, vol. PP, no. 99, pp. 1–12, 2016.

[48] Z. Yuan, Y. Liu, J. Yue, J. Li, and H. Yang, "Coral: Coarse-grained reconfigurable architecture for convolutional neural networks," in *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, July 2017, pp. 1–6.

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

[50] K. Korekado, T. Morie, O. Nomura, H. Ando, T. Nakano, M. Matsugu, and A. Iwata, "A convolutional neural network vlsi for image recognition using merged/mixed analog-digital architecture," in *Knowledge-Based Intelligent Information and Engineering Systems*. Springer, 2003, pp. 169–176.

[51] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "Yodann: An architecture for ultra-low power binary-weight cnn acceleration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2017.

[52] G. Desoli, N. Chawla, T. Boesch, S. p. Singh, E. Guidetti, F. D. Ambroggi, T. Majo, P. Zambotti, M. Ayodhyawasi, H. Singh, and N. Aggarwal, "14.1 a 2.9tops/w deep convolutional neural network soc in fd-soi 28nm for intelligent embedded systems," in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, Feb 2017, pp. 238–239.

[53] B. Moons, R. Uytterhoeven, W. Dehaene, and M. Verhelst, "14.5 envision: A 0.26-to-10tops/w subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28nm fdsoi," in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, Feb 2017, pp. 246–247.

[54] A. Solazzo, E. D. Sozzo, I. D. Rose, M. D. Silvestri, G. C. Durelli, and M. D. Santambrogio, "Hardware design automation of convolutional neural networks," in *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, July 2016, pp. 224–229.

[55] R. DiCecco, G. Lacey, J. Vasiljevic, P. Chow, G. Taylor, and S. Areibi, "Caffeinated fpgas: Fpga framework for convolutional neural networks," in *2016 International Conference on Field-Programmable Technology (FPT)*, Dec 2016, pp. 265–268.

[56] Z. Wang, F. Qiao, Z. Liu, Y. Shan, X. Zhou, L. Luo, and H. Yang, "Optimizing convolutional neural network on fpga under heterogeneous computing framework with opencl," in *2016 IEEE Region 10 Conference (TENCON)*, Nov 2016, pp. 3433–3438.

[57] H. Amin, K. M. Curtis, and B. R. Hayes-Gill, "Piecewise linear approximation applied to nonlinear function of a neural network," *IEE Proceedings - Circuits, Devices and Systems*, vol. 144, no. 6, pp. 313–317, Dec 1997.

[58] E. C. Andre Xian Ming Chang, Berin Martini, "Recurrent neural networks hardware implementation on fpga," in *arXiv preprint arXiv:1511.05552*, 2015.

[59] J. C. Ferreira and J. Fonseca, "An fpga implementation of a long short-term memory neural network," in *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Nov 2016, pp. 1–8.

[60] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang *et al.*, "Ese: Efficient speech recognition engine with sparse lstm on fpga." in *FPGA*, 2017, pp. 75–84.

[61] S. Li, C. Wu, H. Li, B. Li, Y. Wang, and Q. Qiu, "Fpga acceleration of recurrent neural network based language model," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2015, pp. 111–118.

[62] J. A. Renteria-Cedano, L. M. Aguilar-Lobo, J. R. Loo-Yau, and S. Ortega-Cisneros, "Implementation of a narx neural network in a fpga for modeling the inverse characteristics of power amplifiers," in *2014 IEEE 57th International Midwest Symposium on Circuits and Systems (MWSCAS)*, Aug 2014, pp. 209–212.

[63] M. Atencia, H. Boumeridja, G. Joya, F. Garca-Lagos, and F. Sandoval, "Fpga implementation of a systems identification module based upon hopfield networks," *Neurocomputing*, vol. 70, no. 16, pp. 2828 – 2835, 2007, neural Network Applications in Electrical Engineering Selected papers from the 3rd International Work-Conference on Artificial Neural Networks (IWANN 2005).

[64] M. Liang and X. Hu, "Recurrent convolutional neural network for object recognition," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015, pp. 3367–3375.

[65] D. Shin, J. Lee, J. Lee, and H. J. Yoo, "14.2 dnpu: An 8.1tops/w reconfigurable cnn-rnn processor for general-purpose deep neural networks," in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, Feb 2017, pp. 240–241.

[66] Google, "Google tpu alpha," Web site: https://cloud.google.com/tpu/ [Last accessed: 4 September 2017], Sept 2017.