2018 IEEE International Parallel and Distributed Processing Symposium Workshops

Hardware/Software Codesign for Convolutional Neural Networks exploiting Dynamic Partial Reconfiguration on PYNQ

Florian Kästner*, Benedikt Janßen*, Frederik Kautz*, Michael Hübner* and Giulio Corradi†
*Embedded Systems for Information Technology
Ruhr-University Bochum
Bochum, Germany
Email: {Florian.Kaestner, Benedikt.Janssen, Frederik.Kautz, Michael.Huebner}@rub.de
† Xilinx GmbH
Munich, Germany
Email: Giulioc@xilinx.de

***Abstract*—Convolutional Neural Networks (CNN), next to Recurrent Neural Networks (RNNs), are the most important subgroup of Deep Neural Networks (DNNs) due to their recent success in industry, especially regarding image processing. Hardware platforms, like Field Programmable Gate Arrays (FPGAs) and Application Specific Integrated Circuits (ASICs), turned out to be a viable alternative to Graphical Processing Units (GPUs) for DNN implementations, especially according to applications with strict power and performance constraints. Moreover, FPGAs supporting Dynamic Partial Reconfiguration can be reconfigured in the field multiple times. In this paper we present a toolflow designing layer specific hardware blocks and reconfiguring them during runtime. During reconfiguration parts of the inference path are executed in software. Starting at the training of CNNs with the help of the Caffe framework, we provide a parameter parser. Furthermore, layer reimplementation templates in C++ help to easily design layers in software and hardware with the help of High-Level Synthesis according to layer specific characteristics. We further demonstrate the possibilities of our hardware/software codesign approach and hardware reconfiguration on the Deep Landmark Detection application. The whole project is available on Github[1].**

I. Introduction

Convolutional Neural Networks (CNNs) are special type of Deep Neural Networks (DNNs). This kind of neural networks are widely applied to image processing tasks. A major breakthrough in the field of neural network research was the success of a CNN at the ImageNet Large Scale Visual Recognition Challenge in 2012 [2]. Unlike traditional Artificial Neural Networks (ANNs), CNNs are able to process 3-dimensional input data directly. Hence, reshaping of image data is not necessary and not desired due to the loss of information, such as spatial relationships. CNNs consist of heterogeneous layers, such as pooling, fully-connected, convolutional and non-linear activation function layers.

Within the convolutional layers, the input data is processed with 3-dimensional filter-kernels with adaptable weights. The kernel size commonly ranges from 2×2 up to 8×8 and the same depth size as the input data. The filter-kernels are convolved over the input data, each producing a new feature-map containing filter specific features. Due to the fact of the data-flow driven nature, parallel computing is highly beneficial to decrease the execution time of CNNs. Nowadays, the training of DNNs is done with the help of Graphical Processing Units (GPUs). GPUs offer a highly powerful platform with many processing elements, which are programmable with the help of high-level languages such as CUDA or OpenCL. Moreover, frameworks like Caffe [1] further simplify the utilization of GPUs for machine learning applications. However, the advantage in energy efficiency of Field Programmable Gate Arrays (FPGAs) in comparison to GPUs is well known [3],[5],[6]. Furthermore, Nurvitadhi et al. [7] found out, that DNN implementations on FPGAs can outperform their counterpart implementation on GPUs without using batch processing. The energy consumption of the platform is a critical aspect, especially if the target device is mobile and/or the machine learning application cannot be processed on server platforms due to safety constraints. For such applications, like autonomous driving, FPGA or Application Specific Integrated Circuits (ASICs), such as coprocessors, need to be considered.

Within this paper, we present our work on accelerating CNNs on FPGAs with a template hardware/software codesign implementation, and a corresponding toolflow exploiting FPGAs' Dynamic Partial Reconfiguration (DPR) capabilities.

DPR enables the exchange of logic partitions, called Reconfigurable Partitions (RP), within the FPGA fabric. Thereby, the system can be adapted for an optimized execution according to the application, as well as the application phase. This is a major advantage for designing efficient hardware architectures able to adjust the hardware due to characteristics of special types of ANNs, such as CNNs, Recurrent Neural Networks (RNNs) or combinations of both, but also due to the characteristic of layers within these networks. Moreover, the integration of DPR enables the exploration of dynamic adaptations, for instance in growing neural networks or reinforcement learning. However, similar to ASIC development, designing FPGA

[1] https://github.com/KaestnerFlorian/CNN_DPR

978-1-5386-5555-9/18/$31.00 ©2018 IEEE
DOI 10.1109/IPDPSW.2018.00031

implementations is more complex in comparison to software development. Therefore, we developed our toolflow, enabling a semi-automated implementation of hardware/software codesigns.

The main focus of this article is on exploiting DPR to enable accelerating large ANN on small-sized FPGAs, and mitigating long reconfiguration overhead. In addition, we focus on demonstrating the template's capabilities by implementing a deep landmark detection network, that is based on a CNN architecture. The target of this image processing application is to find keypoints within human faces in order to analyze facial expressions and gestures.

Therefore, we design hardware and software Intellectual Property (IP) cores for each convolutional layer used within the target application. The hardware IP cores are implemented with Vivado HLS and adjusted to the characteristics of the layer to increase the efficiency. For the implementation of the hardware/software codesign, we developed the corresponding toolflow, enabling a simplified development of CNNs that are implemented within the Caffe framework. The hardware implementation is based on a DPR overlay architecture.

The paper is organized as follows. Section II describes research activities with similar approaches and the difference to ours. In section III we explain the deep landmark target application and the corresponding neural network. We further present our toolflow in section IV and explain the realization of this with all optimization paths in section V. Section VI presents the results and concludes this research with an outlook.

II. Related Work

CNNs are data-flow driven and offer great possibilities to design hardware accelerator and coprocessor designs exploiting fine and coarse grained parallelism. Due to the recent success of DNNs, especially CNNs for image processing applications, many different research activities are focusing on accelerating the inference phase of these ANNs with the help of FPGA hardware implementations. Most of this research activities focus on designing coprocessor designs using FPGAs as prototyping platforms. Thus, reconfiguration capabilities of FPGAs are mostly neglected.

Sankaradas et al. [8] achieved a performance of 3.37 GMAC per second with a total power consumption of 11 W with a coprocessor design which is capable of accelerating a complete CNN on the Programmable Logic (PL). 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.

Making use of High-Level Synthesis Zhaou et al. [16] implemented a complete five-layer convolutional neural network on a Virtex-7 FPGA. The network, taken from the DeepLearnToolbox, consists of two convolutional layers processing the data with 5×5 filter-kernels and two pooling layers applying average pooling on the results. The necessary data is mainly stored on the on-chip memory of the FPGA using block RAM in first instance and registers in second instance. The reason for this procedure is caused by the two port access to the BRAM hindering further parallelization possibilities of the computations. The design reaches a maximum performance of 3.75 GMAC/s driven by a clock frequency of 150 MHz.

Generally, according to the research of Williams et al. [12], algorithms or functions can be classified being either computational or communication intensive. This roof-line performance model is referred from Zhang et al. [10] to apply this concept performing a design space exploration of beneficial CNN hardware implementations. 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. This research represents the starting point of our hardware implementation because Zhang et al. use this exploration to find layer specific, but global factors applied to every layer type. Our approach exploits the individual characteristics of every layer itself.

A similar approach is outlined by Li et al. [9] investigating the opportunities to customize every type of layer due to their specific characteristics. The major difference is the fact, that Li et al. treat convolutional and fully-connected layers differently in terms of the level of parallelism and memory bandwidth. Furthermore, Li et al. take the limitation of the hardware resources, namely Digital Signal Processing (DSP) and Block RAM (BRAM) units, directly into account. Hence, the 8-layer CNN hardware implementation is working concurrently in a pipeline structure achieving 565.94 GOPs with an energy consumption of 30.2 W on a Virtex 7 device.

Meloni et al. [18] developed an acceleration architecture implementing a highly parallelized hardware convolution engine (HWCE) on a Zynq Z-7045 All Programmable System-on-Chip (SoC). This engine is designed to be reconfigured at runtime to be capable of executing convolutions with different filter-kernel sizes and stride configurations. This reconfiguration as well as activation and controlling is done by two general purpose cores. This cores are also responsible for the scheduling of the convolution tasks as well as for managing data transfers from/to HWCE, and external memory. The scheduling is developed to minimize the bandwidth needed by input and output communication. Working with a frequency of 150 MHz the accelerator reaches a peak performance of 129 GMAC/s.

However, the reconfiguration capabilities of the FPGA are not fully exploited. Our approach offers the possibility to adjust also the cascade structure, line-buffer sizes, partitioning and processing schedule. Therefore, various types of layers, also RNN elements or other generative model approaches, can be added to the design without redesigning previous implementations. For further investigations regarding current hardware implementations of CNNs on FPGAs or ASICs we refer to [4].

III. Deep Landmark Detection

Sun et al. [14] proposed a multi-level regression approach for predicting facial keypoints using a deep CNN cascade.

Facial keypoints are a critical feature for detecting and analyzing faces. With their approach, they address problems occurring with traditional landmark detection methods, which are either trained separately for each keypoint using local regions or directly predicting the keypoints using regressors. While most approaches only roughly predict the initial positions of keypoints, the CNN cascade makes accurate predictions for all keypoints simultaneously already in the first level which are then refined in the levels following. The five different facial keypoints also known as facial landmarks predicted with this approach are: Left eye center (LE), right eye center (RE), nose tip (N), left mouth corner (LM) and right mouth corner (RM). Level one networks are described to be the most complex networks in this approach and therefore represent the main focus. While a deep structure helps to extract global high level features, which are highly non-linear, all level one networks are chosen to be deep and consist of four convolutional layers. The neurons in the lower layers are used to extract local features. While the structure of the level one networks is very common for a CNN, as can be seen in Figure 1, the implementation of some layers own special characteristics, which should me mentioned. First of all, every layer uses *tanh* activation function and absolute value rectification for all neurons. Secondly, the convolutional layers do not share their weights globally, but locally instead. Therefore, in this specific case, the input feature map is divided into four equal parts which are convolved with four different kernel-filter weights depending on the region. The reason for this method is because e.g. eyes and mouths do not share the same high-level features suggested by the authors. Another important fact is that the pooling layers used in this approach are equipped with bias values comparable to convolutional layers.

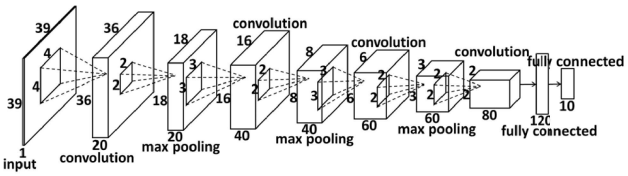

Fig. 1: F1 Network [14]

Based on the approach by Sun et al., Jie Zhang [15] provides a reimplementation of the CNN cascade calling it *Deep Landmark*. This implementation is done using the Caffe framework. For this research we choose the F1 network of this implementation to be accelerated in our hardware/software codesign approach.

IV. Toolflow

Our toolflow enables a simplified hardware/software codesign implementation of CNNs designed with Caffe. Therefore, we developed a parser, which extracts the weights and bias values of a pre-trained Caffe network. The parser expects two files, namely the **.prototxt* file, containing the complete network description, and a **.caffemodel* file, containing the already learned network weights and bias values in a binary format. The parser, written in Python, creates individual text files for the bias and weight values of each layer. In addition, we developed a hardware/software description template for the CNN layers. It simplifies the software and hardware implementation of the CNN's inference path, described in Caffe's **.prototxt* file, together with the weights and bias values extracted by the parser. For CNN software implementation, we support C++. In order to design hardware implementations we utilize Xilinx Vivado HLS 2017.2 to generate layer-specific IP cores. An early verification of the generated IP cores is done via the extracted weights in the testbench. The hardware implementation of the overall system is generated with Xilinx Vivado 2017.2 and based on the HDMI support design for the Xilinx PYNQ-Z1 provided by Xilinx. The software implementation of the overall system is running on the PYNQ Linux image. We visualize the described flow in Figure 2. After designing hardware and software tasks, we are able to dynamically load IP cores into the programmable logic without disturbing the HDMI pipeline, which belongs to the static region of the FPGA. Therefore we achieve a better utilization of its resources, as well as an improved performance.

V. Implementation

The PYNQ-Z1 board contains a Xilinx Zynq-7000 ZC7020. This device consists of a dual-core ARM A9 hard-core processor, equipped with various peripherals, as a Processing System (PS). Moreover, the ZC7020 includes a FPGA, called the Programmable Logic (PL). The PYNQ-Z1 board is further equipped with two HDMI ports. We utilize these HDMI ports to receive the input image and output the resulting one, including the facial landmarks. The HDMI signal pipeline is implemented in the HDMI subsystem inside the PL. A Video Direct Memory Access (VDMA) IP core is used as a framebuffer. Due to the expected clock cycle difference between the HDMI and the CNN pipeline, we separate those using different framebuffers for both subsystems. The input data to CNN hardware IP cores are connected via a simple DMA IP core. Therefore, in contrast to the VDMA, the transmission has to be triggered for every image.

The input image is preprocessed with OpenCV on the PS. In first step, faces are detected, marked with a bounding box and cut out of the image. The resulting partial images, each containing a detected marked face, are resized to 39×39 pixels, and converted into grey scale format. Afterwards, these images are converted into standard arrays. As the last step of the preprocessing, the subtraction of the mean of the pixel values, and the division by the standard deviation of the pixel values are applied to all pixels of the image before it is fed to the CNN.

Table I shows a detailed overview of the F1 CNN described in Section III. As can be seen all convolutional layers own different characteristics. We have to consider three different kernelsizes, namely 2×2, 3×3 and 4×4. We further have to consider different channel dimensions and number of parameters need to be transferred. Our goal is to adjust differ-

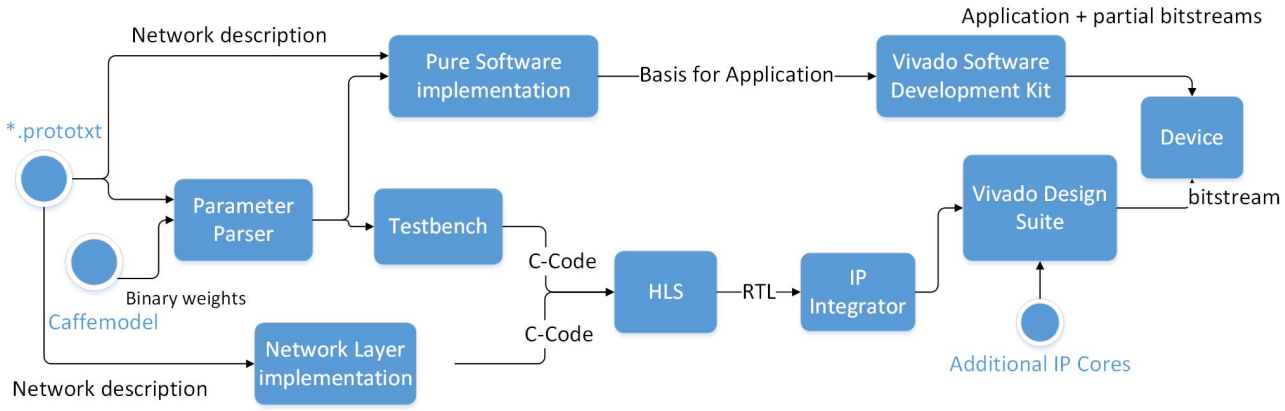

Fig. 2: Flow visualization

ent engineering parameters according to every convolutional layer characteristic in order to reach the best performance considering hardware resource constraints. Shown in Table I, convolution two owns the highest number of computations while convolution four owns the lowest computation to communication ratio. However, these adjustments have to be realized in a time effective manner. Therefore, we choose a high-level design entry with a template implementation, whereof every hardware convolution IP core can be deduced.

A. *High-Level Synthesis*

As outlined in Section I the CNN layers can be divided into three types, namely pooling, convolutional and FC layers. Unlike in convolutional layers, in FC layers the concept of parameter sharing and local connections is not used. As a result these type of layers have a low computation-communication ratio caused by the huge amount of parameters needed for matrix multiplication. Furthermore, newer CNN architectures like GoogleNet [11] successfully replace FC layers with average pooling layers. Therefore, we propose to implement FC layers on the processing system of the SoC in first instance.

Convolutional, activation function and pooling layers are implemented as separate hardware IP cores. According to the reconfiguration flow and sequential structure of the CNN inference path, separated IP cores can be beneficial to use the reconfigurable area in the most effective manner. However, to create hardware implementations accelerating and exploiting characteristics of each layer individually, a high level design entry is beneficial. In order to fast generate valid hardware IP cores we utilize High-Level Synthesis (HLS). An HLS-tool like Vivado HLS from Xilinx [13] produces hardware descriptions at the Register-Transfer-Level (RTL), like VHDL or Verilog, raised from a higher level of abstraction at the algorithm level, like C/C++. Although this process is done mainly automatically, the user can influence the scheduling and allocation process with the help of pragmas and directives.

1) Communication Infrastructure: The communication infrastructure is based on the Advanced eXtensible Interface (AXI) specification. It describes three different types of interfaces. AXI4 Stream supports handshake-based data exchange between a sender and a receiver. It is not memory-mapped, hence there is no addressing overhead. AXI4-Lite is a light-weight memory-mapped protocol, similar to AXI4. The main difference between the two is the support for burst transmissions within AXI4. Each layer is monitored and controlled by the PS over an AXI4-Lite interface.

Due to the data driven flow of CNNs, a high data throughput is needed. Therefore, we utilize AXI4-Stream-based interfaces to transfer input, output and intermediate data. The data transfer of convolutional layers depend on payload data and additional parameters for the processing. Therefore, we utilize a memory-mapped AXI4 interfaces with an adjusted burst size to transport the weight and bias parameters. Due to the weight sharing principle of convolutional layers, the data throughput on this port is much lower compared to the input data port of the layer.

We assume that the incoming data is not reshaped. Hence, feature maps are fed forward in width/height order first and depth order second. In this case the pooling could be applied afterwards in a cascade manner without the need of buffering. In order to buffer the intermediate data in the convolutional layer an input line buffer and an output buffer are used for every convolutional layer except the first one. Due to the single incoming picture depth, the result of the convolution can be directly forwarded to the output stream.

Figure 3 shows the basic principle of the line buffer. The sequential incoming datum via AXI4-Stream is fed to the on-chip memory and shifted to the left when new data is arriving till the end of the line buffer is reached. After reaching the upper left storage the datum is deleted from the memory. The principle scheme can be applied convolving a kernel within a single region. Therefore nine multiplications can be processed in parallel if the needed resources are available and the line buffer implementation allows nine concurrent accesses of the storage. If more convolutions are processed in parallel, the size and the partitioning of the line buffer has to be adjusted.

TABLE I: Detailed layer description of *F1* network [14]

Name	Type	Channels_in	Dimension_in	Kernel Size	Channels_out	Dimension_out	# Operations	# Parameters
Data	data	1	39x39	/	1	39x39	/	/
conv1	Convolution	1	39x39	4x4	20	36x36	414.720 MAC	320
relu1	ReLu	1	39x39	/	20	36x36	25.920 comp	/
pool1	Pooling	20	36x36	2x2	20	18x18	25.920 comp	/
conv2	Convolution	20	18x18	3x3	40	16x16	1.843.200 MAC	7.200
relu2	ReLu	40	16x16	/	40	16x16	10.240 comp	/
pool2	Pooling	40	16x16	2x2	40	8x8	10.240 comp	/
conv3	Convolution	40	8x8	3x3	60	6x6	777.600 MAC	21.600
relu3	ReLu	60	6x6	/	60	6x6	2.160 comp	/
pool3	Pooling	60	6x6	2x2	60	3x3	2.160 comp	/
conv4	Convolution	60	3x3	2x2	80	2x2	76.800 MAC	19.200
relu4	ReLu	80	2x2	/	80	2x2	320 comp	/
pool3_flat	Flatten	60	3x3	/	540	1x1	/	/
conv4_flat	Flatten	80	2x2	/	320	1x1	/	/
concat	Concat	860	1x1	/	860	1x1	/	/
fc1	InnerProduct	860	1x1	/	120	1x1	103.200 MAC	103.200
relu_fc1	ReLu	120	1x1	/	120	1x1	120 comp	/
fc2	InnerProduct	120	1x1	/	10	1x1	1.200 MAC	1.200
relu_fc2	ReLu	10	1x1	/	10	1x1	10 comp	/
TOTAL							3.220.000 MAC 77.0900 comp	152.720

Increasing the size of the line buffer by one element offers the opportunity to apply sixteen concurrent multiplications. By default, Vivado HLS implement this line buffer into a dual port block RAM (BRAM) or look-up-table RAM (LUTRAM) instance preventing further parallelism due to the limited access. In order to increase the accessibility, the pragma *array partition* to the line buffer variable in the C/C++-code can be applied. This pragma offers the opportunity to generate logical connected, but physical separated memory instances either in a block packaged or cyclic packaged manner. Although, this results in a higher resource utilization, we are able to adjust the accessibility of buffers due to specific implementation characteristics.

2) IP-Core Optimization: The convolution is the most computational intensive layer in CNNs. Listing 1 shows the pseudocode for a convolutional layer. The kernels are convolved over the whole input width, height and depth, with specific stride and padding properties. By default all nested loops are executed in a sequential manner. Within this convolution process Vivado HLS provide different pragmas to influence the scheduling and resource allocation.

Two of the most important pragmas used to optimize the scheduling of loop execution are *unrolling* and *pipelining*. Therefore, we are able to adjust the loop scheduling and thus allowing the amount of operations to be processed in a parallel or in a pipelining manner. Due to the limited resources given by the size of the reconfigurable area, the loop scheduling, operation allocations, line buffer size and partitioning representing engineering factors. Thus, the goal of the individual high level design space exploration is to find the best trade-off between resource utilization and performance for every layer under the given resource limitation and infrastructure. According to the limited on-chip resources we propose to

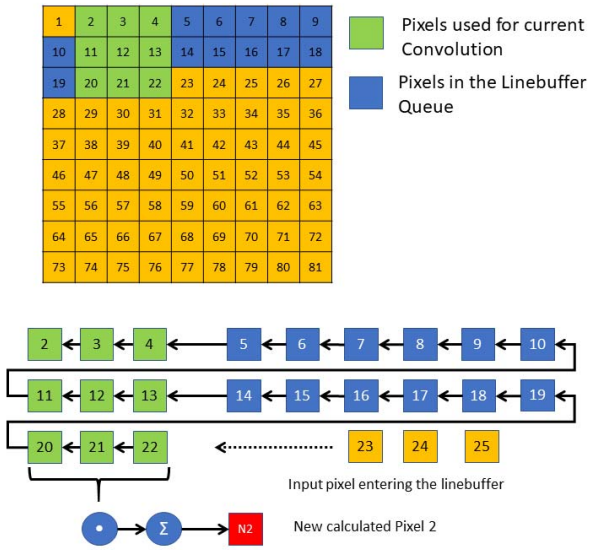

Fig. 3: Input line buffer scheme

pipeline *Loop4* with the rewind property for every convolution IP core. As a result, *Loop5* and *Loop6* are automatically unrolled. Pipelining forces the scheduler to share resources while minimizing latency. Within this structure, the amount of parallel operations able to be executed in parallel dependes on the kernelsize. Thus, we apply an additional *unroll* pragma with an individual factor determined by the kernelsize for *Loop4*.

Furthermore, allocation pragmas are used to force the scheduler taking resource limitations into account. This allows the hardware programmer to specify the number of operations

TABLE II: Results of the HLS IP-core optimization for every convolutional layer

Name	Resource Utilization(%)				f(MHz)	Latency	Exec_hw(ms)	GMAC_hw(s)	Exec_sw(ms)	speedup
	BRAM (%)	DSP48E (%)	FF (%)	LUT (%)						
conv1	24	29	24	34	100	265817	2.7	0.154	58.04	21
conv2	21	20	17	31	100	533989	5.3	0.348	245.52	46
conv3	21	20	14	28	100	230385	2.3	0.338	103.34	45
conv4	24	7	9	18	100	43794	0.4	0.179	10.85	27

implemented in hardware. However, applying resource sharing also has big disadvantages regarding scheduling, routing and placing process. Due to the recurring usage of the same hardware components, the maximum frequency increases. This issue is especially critical if BRAM resources are involved. To ease this affect we add output and input registers to each BRAM instance and increase the clock uncertainty to met the timing of a *10ns* clock period through every IP-core.

To further increase the performance and decrease resource utilization we use a simple quantification of floating-point operations. Therefore a datatype with a bit-width of 32 bit including eight integer bit-width length, is used for every operation and casted from the floating-point values. The negative effect of this quantification regarding the accuracy is measurable, but below 1 %. We propose to not further lower the precision of the parameters, and inputs due to the regression task characteristic, where accuracy loss is more strongly apparent than in classification tasks. However, even this simple quantification improves the performance of every convolution IP-core by factor 2. The results of the hardware implementation optimization for every convolution layer are shown in Table II.

Listing 1: Pesudo-code Convolution Layer

```
Loop1:for filterNr in 1..filterNum // featuremap
    depth
  Loop2:for pictureX in 1..pictureSizeX // picture
      width
    Loop3:for pictureY in 1..pictureSizeY // picture
        height
      Loop4:for depth in 1..depthSize // Output
          featuremap depth
        Loop5:for filterx in 1..filterSizeX //
            Filter x direction
          Loop6:for filtery in 1..filterSizeY //
              Filter y direction
            xn = picturex + filterx
            yn = picturey + filtery
            output += weights(filterx,filtery,depth,
                filterNr)*input(xn,yn,depth)
          end
        end
      end
    end
  end
end
```

We consider the following area constraints for our hardware design due to the limitation of the reconfigurable area: A maximum of 30 % of Block RAM (BRAM), 35 % of Digital Signal Processor Instances (DSP48E), 28 % of Flip-Flops (FF) and 40 % of Look-up-tables (LUT).

The remaining logic of the PL is used by the HDMI infrastructure. As expected, convolution two takes most time to compute all dot-products due to the high amount of computations needed. Different communication to computation ratios of the convolution layers are defining different optimization criteria and therefore inhibit to equal the performance to the same level. As well as for the hardware implementation of the convolution layers, a line buffer is used to buffer the incoming data for the pooling layers. Depending on input dimensions and the kernelsize property of the pooling layer, the line buffer is partitioned into subarrays, where the amount of subarrays is equal to the kernelheight. Within each of these subarrays the maximum value can be determined in parallel. The outgoing value is the maximum of the result of all subarrays in a pipelining manner. The amount of line buffer shift operations are depending on the stride property of the pooling layer. Unlike convolution and pooling hardware IP-cores, the activation function IP-core is working with the data stream without buffering. The floating-point *max* function used for pooling and ReLu cores is implemented applying a function instance provided by Vivado HLS math library optimized for hardware implementations. Thus we archived an interval and latency of one clock cycle from arriving input to dispatched output for both IP-core types while minimizing area utilization. However, this additional IP cores are not used in later implementation due to the research in section V-B, but are also provided in the Github repository.

B. Dynamic Partial Reconfiguration

In this section, we will present the DPR capabilities of our system. As shown in the previously, the hardware implementations of the CNN layers require more resources than offered by the target device. Moreover, the resources are shared between the CNN hardware and the HDMI subsystem. Therefore, we applied DPR in order to share FPGA resources between multiple layers of the CNN. Reconfiguring a FPGAs is time consuming. Even with DPR, which enables a reduction of the timing overhead due to a reduction in the amount of reconfigured logic, the reconfiguration time is still a strong constraint. Beside the timing constraint, there are also area constraints. The relocation of partial bitstreams, and the modification of reconfigurable partitions (RPs) in the field is not supported. Hence, the size of the RP has to be chosen with respect to all hardware modules targeting it. The size and placement of the RP for our CNN framework is shown in Figure 4. We chose to implement a single RP to cope with the size of the convolution layers. The contained resources, as well as the resources used by the convolution layers are compared in Table III. In order to allow the hardware acceleration of multiple layers, layers are swapped in and out the RP, according to the data-flow

inside the CNN. This process is done without affecting the remaining logic of the HDMI subsystem.

The RP's placement is done over two horizontal clock regions. During evaluation we discovered several issues using RP over two or more vertical clock regions due to timing constraints. Another constraint is the high resources usage of the static logic, mainly determined by the HDMI subsystem, and the used I/O pins. We could not use the complete clock regions, indicated by the margin on the right side of the RP in Figure 4. The reason for this is the I/O pin placing of the HDMI subsystem on the right side. An additional issue which arises by applying reconfiguration is the reset of the reconfigured programmable logic. To avoid this we propose to use a mechanism to manually reset the IP cores after reconfiguration. In this research a simple General Purpose Input/Output (GPIO) IP core is used with a reset active low characteristic. The resulting partial bitstreams have all the same size of 2.1 MB. This represents approximately half of the total bitstream size of 4.0 MB including the static and partial areas. For further information about how to use reconfiguration inside Vivado including corresponding Tcl commands we refer to [17].

TABLE III: Available resources on the Zynq XC7Z020, and in the RP, in comparison to used resources by convolution layer RMs.

Resource	Number of Resources					
	XC7Z020	RP	conv1	conv2	conv3	conv4
LUT	53200	18800	18163	16948	15070	9981
FF	106400	37600	26436	18351	15627	9746
BRAM	280	80	68	59	59	69
DSP48E	220	80	64	45	45	16

C. Results

After the generation of all bitstreams, the overall system has to be coordinated and controlled via the PS. Input and output memory spaces are allocated by our application with the help of a Python script. This script utilize the xlnk-driver in order to reserve virtual memory. This virtual memory is later mapped to the physical space to realize the communication with the PL, including framebuffers, memory for all parameters and control registers of the IP core interface. Hence, the incoming pictures from the HDMI framebuffer are extracted, preprocessed with the help of OpenCV and fed to the desired physical address of the CNN-DMA read memory space. Thereafter, the IP core is configured, weights and bias parameters of the layers are extracted from the .txt files, shifted to the desired DMA memory and the IP core is started. Then, the DMA is triggered to transfer the given amount of data depending on the input dimensions of the layer. The concurrently streamed output of the IP core is fed back to the DMA and written to the DDR memory. As soon as the data transfer is accomplished, we reconfigure the IP core with the following one. In order to apply DPR, we use the PCAP interface and the corresponding driver. Thus, we are able to reconfigure the PL through software commands. The measurement of the reconfiguration

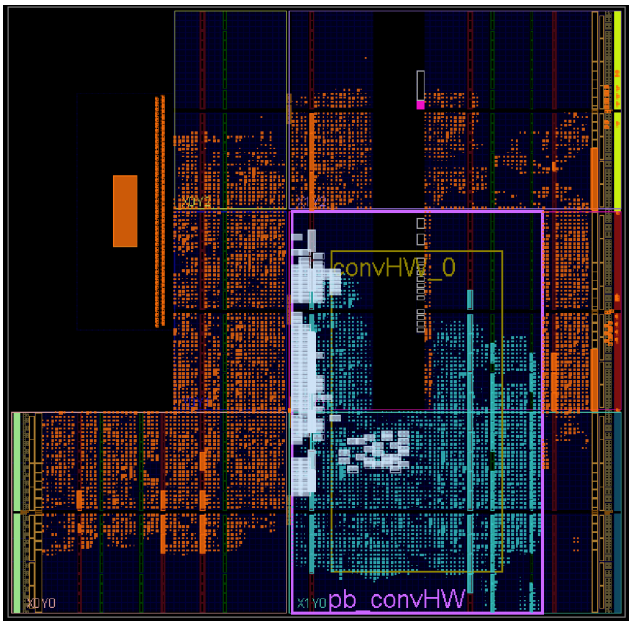

Fig. 4: Placement of the static logic (orange), the reconfigurable area (blue) and the fixed ports (white)

times for every convolutional layer are listed in Table IV. As

TABLE IV: Reconfiguration time of every convolutional layer compared to their execution time

Name	Reconf_time(ms)	Exec_time(ms)	Exec/Reconf_ratio
conv1	155	2.7	0.0174
conv2	123	5.3	0.0431
conv3	156	2.3	0.0147
conv4	116	0.4	0.0034

can be seen in Table IV, the reconfiguration time overhead is surprisingly way too high compared to the actual execution time of the hardware implementations. All in all, the sequential reconfiguration of convolutional layers would cause a higher execution time of the CNN inference path then processed as a pure software solution on the dual ARM core. Therefore, we propose to use a combination of hardware/software processing. Pooling and activation function layers are executed in software. Additionally, due to the high reconfiguration times, we alternate software and hardware executions of convolutional layers. As we use Linux as an operating system, pthreads are beneficial in order to use the reconfiguration time while keeping the processing pipeline. Therefore, we start a new pthread to dynamically reconfigure the hardware and join this thread after all software processing tasks of the CNN inference path are done. To evaluate this approach we measured the execution time of ten CNN inference runs of ten different pictures with two versions of alternating hardware/software executions and compared them against the pure software execution as can be seen in Table V. The mean execution time of the pure software solution is 0.4458177s. If convolution one and three are outsourced to the PL a mean speedup of 1.54

compared to the software solution is achieved. Otherwise, if convolution two and four are executed in hardware, the overall execution time is reduced by 246.5ms leading to the best achieved speedup of 2.24.

TABLE V: Evaluation of the hardware/software codesign with two Versions of alternating hardware/software layer executions

run	exec_t(s)_conv1/3	exec_t(s)_conv2/4)	exec_t(s)_sw
1	0.297087	0.199641	0.445513
2	0.296633	0.199324	0.445756
3	0.297120	0.199483	0.446422
4	0.296687	0.199305	0.445975
5	0.297026	0.199341	0.445540
6	0.297017	0.199112	0.446063
7	0.296937	0.199425	0.445755
8	0.296765	0.199027	0.445863
9	0.297262	0.199278	0.445659
10	0.297242	0.199633	0.445631
ave	0.296977	0.199356	0.445819

VI. Conclusion and Outlook

In this paper we propose a template-based hardware/software codesign approach that enables the exploitation of DPR to accelerate CNNs. As an example application a deep landmark CNN was chosen. The proposed toolflow enables a fast design space exploration, beginning at a high level description of the network in Caffe. Hence, we develop a parser to extract all weight and bias data to further use them in an embedded system environment. We enable the use the same description language for hardware and software layers enabling a fast design space exploration. Therefore, the inference path of typical CNN layers, convolutional layers in particular, are reimplemented in C++.

The hardware IP cores are developed with the help of Vivado HLS. We optimize every layer due to their characteristic using HLS directives and pragmas as engineering factors. The results of the optimization are comparable with those described in section II, assuming a linear increase of performance depending on hardware utilization. The static bitstream, including HDMI subsystem, and the partial bitstreams are generated according to the DPR-flow from Xilinx. To use the reconfiguration time effectively, we decided to use an alternating execution of CNN layers in hardware and software.

This research evaluates the most disadvantageous reconfiguration level. However, we achieved a 2.24 times faster execution compared to the pure software solution. With our template hardware/software codesign approach much larger designs could be designed to implement larger and more layers in hardware and reconfigure the whole network instead. Hence, the development and design space exploration of such designs are eased with the help of this research. All project sources are available in Github.

References

[1] Jia, Yangqing and Shelhamer, Evan and Donahue, Jeff and Karayev, Sergey and Long, Jonathan and Girshick, Ross and Guadarrama, Sergio and Darrell, Trevo, "Caffe: Convolutional Architecture for Fast Feature Embedding", arXiv preprint arXiv:1408.5093, 2014

[2] ImageNet Classification with Deep Convolutional Neural Networks, Alex Krizhevsky and Sutskever, Ilya and Hinton, Geoffrey E, Advances in Neural Information Processing Systems 25, 1097–1105,Curran Associates Inc.,2012

[3] 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. 230235.

[4] Javier Hoffmann, Osvaldo Navarro, Florian Kastner, Benedikt Janen, Michael Huebner, Chair for Embedded Systems of Information Technology,Ruhr-Universitat Bochum, "A Survey on CNN and RNN Implementations", PESARO 2017 : The Seventh International Conference on Performance, Safety and Robustness in Complex Systems and Applications

[5] M. Al Kadi, B. Janssen, and M. Huebner, Fgpu: An simt-architecture for fpgas, in Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ser. FPGA 16. New York, NY, USA: ACM, 2016, pp. 254263. [Online]. Available: http://doi.acm.org/10.1145/2847263.2847273

[6] 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. 14.

[7] 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. 14.

[8] 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. 5360.

[9] 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. 19.

[10] 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. 161170. [Online]. Available: http://doi.acm.org/10.1145/2684746.2689060

[11] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, Going deeper with convolutions, CoRR, vol. abs/1409.4842, 2014. [Online]. Available: http://arxiv.org/abs/1409.4842

[12] S. Williams, A. Waterman, and D. Patterson, Roofline: An insightful visual performance model for multicore architectures, Commun. ACM, vol. 52, no. 4, pp. 6576, Apr. 2009. [Online]. Available: http://doi.acm.org/10.1145/1498765.1498785

[13] Xilinx Vivado Design Suite, User Guide High-Level Synthesis, UG902 (v2017.2) April 5, 2017

[14] Y. Sun, X. Wang, and X. Tang, Deep convolutional network cascade for facial point detection, Chinese University of Hong Kong and Chinese Academy of Sciences, 2013

[15] J. Zhang, Predict facial landmarks with deep cnns powered by caffe, Huazhong University of Science and Technology China, Jan. 2016. [Online]. Available: https://github.com/luoyetx/deep-landmark

[16] 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.

[17] Xilinx Vivado Design Suite User Guide Partial Reconfiguration, UG909 (v2017.2) April 5, 2017

[18] P. Meloni, G. Deriu, F. Conti, I. Loi, L. Raffo and L. Benini, "A high-efficiency runtime reconfigurable IP for CNN acceleration on a mid-range all-programmable SoC," 2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig), Cancun, 2016, pp. 1-8. [Online]. Available: doi: 10.1109/ReConFig.2016.7857144