2016 IEEE Computer Society Annual Symposium on VLSI

Hardware Design Automation of Convolutional Neural Networks

Andrea Solazzo[1], Emanuele Del Sozzo[1], Irene De Rose[1], Matteo De Silvestri[1], Gianluca C. Durelli[1], Marco D. Santambrogio[1]

[1]Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria, Milano, Italy,
{andrea.solazzo, irene.derose, matteo.desilvestri}@mail.polimi.it,
{emanuele.delsozzo, gianlucacarlo.durelli, marco.santambrogio}@polimi.it

***Abstract*—Convolutional Neural Networks (CNNs) are a variation of feed-forward Neural Networks inspired by the biological process in the visual cortex of animals. The interest in this supervised learning algorithm has rapidly grown in many fields like image and video recognition and natural language processing. Nowadays they have become the state of the art in various applications like mobile robot vision, video surveillance and Big Data analytics. The specific computation pattern of CNNs results to be highly suitable for hardware acceleration, in fact different types of accelerators have been proposed based on GPU, Field Programmable Gate Array (FPGA) and ASIC. In particular, in the embedded systems context, due to real time and power consumption challenges, it is crucial to find the right tradeoff between performance, energy efficiency, fast development round and cost. This work proposes a framework meant as a tool for the user to accelerate and simplify the design and the implementation of CNNs on FPGAs by leveraging High Level Synthesis, still providing a certain level of customization of the hardware design.**

***Index Terms*—Field Programmable Gate Arrays, Convolutional Neural Networks, Design methodology, Heterogeneous MPSoCs**

I. Introduction

CNNs [1, 2] are currently the state-of-the-art approach for image recognition and classification, belonging to the class of Supervised Learning. The idea behind the CNN algorithm is to replicate the mechanism in the primary visual cortex of living organisms; in particular, these cells are arranged in *receptive fields* that capture information from local regions of the field of view. This is reflected in a sparse connectivity of the artificial neurons (called *perceptrons*), which differs from the fully-connected layers of standard Aritificial Neural Networks (ANNs).

Thanks to their flexibility and effectiveness, CNNs are widely applied in different fields, like computer vision and Big Data analysis [3, 4, 5], where high performance are required. For this reason, many research works focused on hardware accelerating the classification phase of CNNs. Indeed, the dataflow computational pattern of CNN, along with the almost complete absence of control structures, make such algorithm suitable for hardware acceleration. Different works on Graphic Processing Units (GPUs), FPGAs, and Application Specific Integrated Circuits (ASICs) [6, 7, 8] proved the efficiency of such approach in terms of throughput.

In this work, we chose FPGAs as target device for the hardware acceleration of CNNs. This choice was driven by different factors. First of all, in the context of CNNs, finding the optimal model to solve a given problem requires several attempts, as well as analyzing the design space due to the number of degrees of freedom available in a CNN. This fact could be a problem in case of an ASIC, where the different attempts should be made only by simulating the system. On the other hand, taking advantage of its reconfigurability, FPGA offers the chance to directly test the network on the device, matching consumption and performance to find the best solution. Second, the FPGAs provide a better performance per watt trade-off with respect to the GPUs, where, despite the high performance, the power consumption represents the main issue, in particular for the embedded context.

One of the main issue of FPGA design is the steep learning curve required to target this design and the long development time needed to generate working solutions. To mitigate this problem a wide range of Computer Aided Design (CAD) tools have been proposed over the years and our research group realized one tool [9] targeting specifically the generation and synthesis of CNN for Xilinx FPGAs. Our solution, starting from the weights of an already-trained network, produces a synthesizable C++ code and the scripts for *Vivado HLS* [10] and *Vivado Design Suite* [11] in order to automatize the implementation on FPGA starting from the High Level Synthesis (HLS) process to the bitstream generation

This paper presents an extension of our framework focusing on the estimation of resource usage of the CNN under development. Such improvement will allow the designer to have a clear idea about the resources used by its implementation without the need of doing any synthesis, effectively reducing the development time. The work presented here allows to attain a good prediction accuracy for the usage of different resources available on the FPGA, allowing for a prediction error of less than 30 BRAMs, 800 LUTs, and 1400 FFs, for most of the our test cases.

The paper is organized as follows. Section II discusses related work and Section III gives an overview on CNNs. Section IV presents the proposed framework, while Section V explains the resource estimation model. In Section VI we provide the experimental results obtained from the estimation of different networks implementations. Finally, Section VII contains conclusions and future works.

II. Related Work

In the late 1990s, Professor Yann LeCun proposed the concept of CNNs, whose original purpose was the recognition of handwritten digits and letters [1]. Thanks to their capability of identifying different and complex features in images, nowadays CNNs are one of the most used and efficient supervised learning techniques applied to image classification [5]. Moreover, CNNs proved their effectiveness in other fields; indeed, authors in [3] applied a 3D CNN model for human action recognition, whereas the work presented in [12] reports a CNN-based natural language processing engine.

As the size of the network grows, the computational load of the CNN highly increases as well. For this reason, many works in literature exploited the dataflow nature of CNNs classification phase and proposed hardware accelerations on GPUs [5], FPGAs [8, 13, 14] and even ASICs [15]. Besides, although the training phase is more suitable for a software implementation due to its control dependent nature, hardware implementations of such phase on both GPU [16, 17, 18] and FPGA [19] produced interesting results.

The work presented in [8] represents the state of the art for the acceleration of CNN convolutional layers on FPGA. The authors based their design space exploration on the *roofline model* [20], i.e. a model designed to produce performance estimation based on both performance peak and off-chip memory bandwidth. As result, the authors presented an implementation of the popular AlexNet CNN [5] that surpassed the previous works in literature by achieving 61.62 GFLOPS (as peak performance) at a working frequency below 100MHz.

Farabet et al. presented a programmable CNN processor implemented on a low-end DSP-oriented FPGA [13]. The proposed processor is designed support a vector instruction set that matches the elementary operations of a CNN. On the other hand, the authors developed a `Lush`-based network compiler software that, starting from a `Lush` description of the CNN, compiles a sequence of instructions for the CNN processor. This work was demonstrated for a real-time face detection, but it could be employed as a lightweight embedded vision system for mobile robots.

Peemen et al. exploited computation reordering and local buffer usage in order to improve throughput and reduce energy consumption of CNN hardware accelerators [14]. At the same time, the authors introduced a new analytical methodology that relies on loop transformations in order to optimize nested loops for inter-tile data reuse. Experimental results demonstrated a reduction in data movement up to 2.1X and a remarkable boost in MicroBlaze soft-core performance.

Differently from the works available in literature, the proposed framework offers the capability of easily generating a synthesizable CNN for FPGA acceleration, starting from the network weights. As result, this work may significantly reduce design time and increase productivity, since it avoids the designer the effort of refining the code in a HLS compliant way.

III. Convolutional Neuronal Networks

CNNs have a specific structure divided in two main blocks: the convolutional and the linear parts. The first one distinguishes them from classical ANNs and it is the key point of those networks. Indeed, convolutional layers are responsible of the extraction of *features* from the input images. The *feature extractor* is made of an arbitrary number of layers, usually alternated with sub-sampling layers.

The second block is a fully-connected neural network (also known as *Multi-Layer Perceptron (MLP)*) that is used to collect the extracted information and classify the input image. In the following paragraphs we discuss in details the different types of layers focusing on the computation flow and re-sizing of the data in the feed-forward process.

Convolution Layers: The key components of CNNs are the weighted filters (kernels) that compose the convolutional layers. The weights of the kernels are defined during the training phase, in which the prediction error with respect to each weight is calculated and *back-propagated* using an algorithm such as Stochastic Gradient Descent (SGD). At high level of abstraction, the feature maps, output of each kernel, are obtained swiping the kernel across the image. Then, the results of each layer become the input of the next layer. The dimensions of a feature map will be reduced from the original image according to the dimensions of the applied filter:

$$width_{new} = width_{old} - width_{kernel} + 1 \tag{1}$$

$$height_{new} = height_{old} - height_{kernel} + 1 \tag{2}$$

Sub-sampling layers Since CNNs need to process large amounts of data, it is useful to find a way to separate relevant information from the others. For this reason convolution layers are commonly alternated to sub-sampling ones. Sub-sampling layers have the role to reduce significantly the amount of data forwarded through the network. Basically, the idea is the same as for convolutional layers, at an high level of abstraction a filter is swiped across the image, reducing clusters of pixels to a single value. In *Mean-pooling*, as an example, the kernel calculates the average value of the pixels, creating a point for the new image; another type of sub-sampling is the *Max-pooling*, which returns the maximum value present in the filter matrix. The dimensions of the output of the sub-sampling layers can be calculated similarly to the convolution layers by the equations 34, where p_{step} is the amplitude of the shift of pooling kernels.

$$width_{new} = \left\lfloor \frac{width_{old} - width_{kernel}}{p_{step}} \right\rfloor + 1 \tag{3}$$

$$height_{new} = \left\lfloor \frac{height_{old} - height_{kernel}}{p_{step}} \right\rfloor + 1 \tag{4}$$

Linear layers Linear layers are located after the convolutional part and they are responsible of the classification process. Their functional unit is the *perceptron*, a simple artificial neuron that calculates a weighted sum of the inputs. The output of each linear layer is computed by equation 5.

$$o_j = \sum_{i=0}^{I}(w_i \cdot x_i) + b_j \tag{5}$$

The last layer of the linear part of a CNN has as many neurons as the number of classes to be recognized and it can be followed by the *LogSoftMax* operator (equation 6), which normalizes the output vector **z** into a set of normalized values corresponding to the probability of the image to belong to a certain class.

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad for\ j = 1, \ldots, K \tag{6}$$

IV. Framework Overview

In this work we present a web-based framework that allows to design and customize a CNN by means of a web-based Graphical User Interface (GUI). The client-side has been implemented in HTML5 and Javascript, while the back-end is written in Python. The workflow of the application is shown in Fig. 1. The framework requires a high level specification of the network and a file containing the trained weights as inputs. In the following subsections the different phases of the workflow are presented, while a full description of the framework can be found in [9].

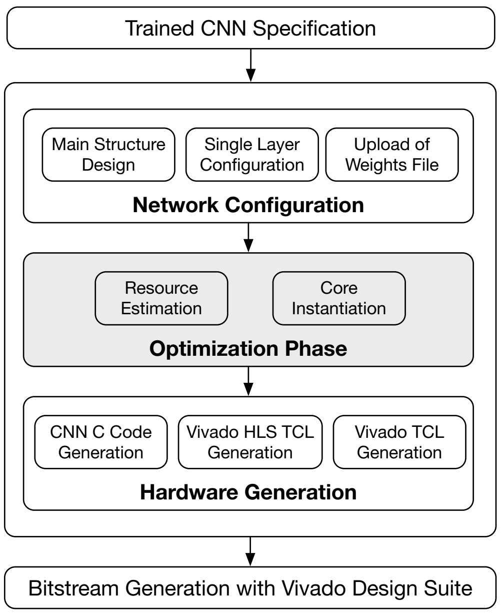


Fig. 1: Workflow of our framework: the part highlighted corresponds to the contribution described in this paper.

A. Network configuration

The application provides to the user a simple interface to going through the design process. As it can be seen from Fig. 1, the user must specify the main structure of the CNN in the first place, by providing the number of convolutional and linear layers, and the dimension of the input data that the network will process. Another input required to generate the CNN code is the set of weights of the different layers. The user can decide either to upload his own file, which can be exported with little effort from Machine Learning frameworks available online (e.g. Torch [21]). With respect to the previous version, it is also possible to let the application to generate random values for the weights, so that he can evaluate the hardware implementation of the network focusing on performance and resources. Then, by mean of a graphical representation of the network, each layer can be customized by specifying its parameters. For convolutional layers, the user can set the number of filters and their dimension. Moreover, a sub-sampling layer implementing Max- or Mean-pooling can be included and configured at this stage. For linear layers, the user can specify the number of neurons in the network.

B. Optimization phase

Once the network configuration has been completed, the application provides a report of the hardware resources (i.e. DSPs, BRAMs, FFs and LUTs) that would be used by the CNN IP core on the selected target device. The technique used to estimate the resources utilization is the main contribution of this work and will be discussed in the next chapter.

Given the overview of the occupied resources, the user can then choose how many cores to instantiate in the final hardware design. Each of those cores will implement the whole network in hardware. In this way it would be possible to parallelize and speed up the overall computation by dividing the images to be processed among the different cores. At this point, the back-end of the application will generate the C++ source code implementing the CNN and the tcl scripts needed by the Xilinx Tools as output.

C. Hardware design

At this point, the back-end of the application will generate the C++ source code implementing the CNN and the tcl scripts needed by the Xilinx Tools as output. The tcl scripts are used to synthesize the whole design up to the generation of the bitstream with Vivado and Vivado HLS. As regards to the HLS of the CNN core, we exploited the dataflow pattern of the algorithm and we added directives to the C++ source code generated by the framework (such as loop pipelining in the most nested loops of the convolutional part). Since networks of different dimensions and configurations can be optimized with case-specific approach, we did not introduce optimizations that may constrain the design to a fixed network pattern. However, it is always possible for the user to re-synthesize his own custom core by using the provided code as starting point.

V. Resource Estimation

This section describes the technique used to allow the framework to estimate resource usages of the CNN under development. As explained in Section IV the availability of these estimations allows the designer to instantiate multiple CNNs on the reconfigurable fabric that can be used in parallel.

The translation of the CNN functionality in HDL is done using Vivado HLS, which already provides resource estimation by itself. However, this estimation comes only after the synthesis phase and as a consequence waiting for it might greatly impact development time. Our idea is to provide to the user an estimation of the resource utilization without the need to go through the HLS phase, thus speeding up the design phase. In other words we want to identify a function F that given the description of a CNN will provide us an estimation of its resource usages in terms of Block RAMs (BRAMs), Digital

TABLE I: Parameters Identifying the CNN for each of its stages: Input image, Convolutional layers, Linear layers, and Classification and Prediction layer.

Input	**Conv. Layer**	**Linear Layer**	**Class. Layer**
Image Width	Kernel Width	Inputs	Classes
Image Height	Kernel Height	Neurons	
Color Depth	Feature Maps In		
	Feature Maps Out		

Signal Processors (DSPs), Look-Up Tables (LUTs), and Flip Flops (FFs). F is then defined as: $F : CNN_{Set} \rightarrow \mathbb{N}^4$, where CNN_{Set} identifies the set of all the possible CNNs.

To identify F there are two possible options. The first one is to perform a deep analysis of the code given as input to the HLS phase and determine how each construct impacts on the final resource usage. The second one instead tackles the problem from the opposite perspective, aiming at identifying the model by interpolating information coming from the synthesis of networks with well known parameters.

The structure of CNN code is regular enough to allow for the first hypothesis, however such method is too bounded to the specific HLS tool considered, and the optimizations that are introduced, automatically or upon request. For this reason we decided to follow the second method stated above and doing a data-driven exploration.

Performing a data-driven exploration means that we have to start by synthesizing a certain amount of networks and then interpolate the obtained data. Ideally the initial data should be a representative sample of the whole multidimensional space that can characterize a given CNN. A summarization of the parameters of a given CNN is given in Table I. As an example, if we look at the parameters needed to characterize a CNN having two convolutional layers and two linear layers, we need to explore a 16-dimensional space: 3 for the input image, 4 for each of the convolutional layers, 2 for each of the linear layers, and 1 for the classification and prediction stage. So our F function is a function $\mathbb{N}^{16} \rightarrow \mathbb{N}^4$ if we consider networks composed only of one convolutional and one linear layer.

Clearly, sub-sampling a 16-dimensional space is a cumbersome task and will require a huge number of synthesis, possibly impairing the benefit of the availability of the models. We decided to break down the problem in different sub-problem and more specifically we started from the assumption that each convolutional, linear, and classification layers can be analyzed on its own and then resource usage of every layer can be summed up to obtain the overall resource utilization of the network. A generic $CNN \in CNN_{Set}$ is then defined as: $CNN = \{CONV, LINEAR, Classes\}$

where $CONV$ and $LINEAR$ are the set of all the convolutional and linear layers present in the CNN. Note that from this description we removed the characteristics of the input image since their values are directly correlated to other parameters in the convolutional and linear layers and are so redundant in the description. This allows to break down the F function as follows:

$$F(CNN) = \sum_{c \in CONV} F_{Conv}(c) + \sum_{l \in LINEAR} F_{Linear}(l) + F_{Class}(Classes)$$

Furthermore, we assume to be possible to predict each of the resources (BRAM, DSP, LUT, FF) independently. So the functions F_{Conv}, F_{Linear}, and F_{Class} are a composition of 4 different functions each one predicting the resource usage of a specific resource.

At this point the initial problem of finding a mapping from a 16-dimensional to a 4-dimensional space has been break down into multiple simpler problems that are more affordable and which greatly reduce the number of synthesis needed to sample significant points in the design space.

A. Convolutional Layer Estimation

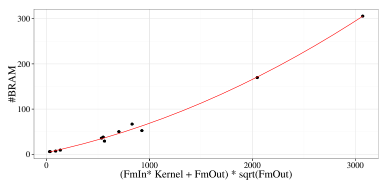
In order to estimate the resource usage of a convolutional layer we perform few synthesis of different layers considering a sub-set of the possible input parameters. In particular we used squared kernels of dimension ($Kernel$) ranging from 2 to 5, input feature maps ($FmIn$) in the range 1 to 64, and output feature maps ($FmOut$) ranging from 8 to 64.

Starting from the data collected from these synthesis we fitted four different linear models to predict the different resource usage. We did multiple projection of the 4-dimensional data in a 2-dimensional space and applying transformations to input parameters to see if non-linearity in the model could be hidden by combining and applying non linear function to the input parameters. The best models we found are presented here below, while Figure 2 shows the prediction done by the model for the convolutional layers. The major thing to note here is that the number of DSP is practically constant in the convolutional layers.

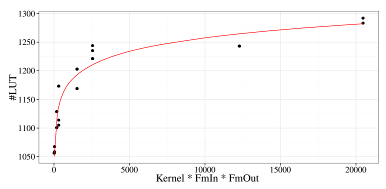
$$\begin{aligned} X_{BRAM} &= (FmIn * Kernel + FmOut) * \sqrt{FmOut} \\ F_{Conv,BRAM} &= 3.86 + \\ &+ 4.864 * 10^{-2} * X_{BRAM} + \\ &+ 1.613 * 10^{-05} * X_{BRAM}{}^2 \\ F_{Conv,DSP} &= 5 \\ F_{Conv,LUT} &= 941.79 + \\ &+ 34.28 * log(Kernel * FmIn * FmOut) \\ F_{Conv,FF} &= 685.84 + 40.92 * log(Kernel + FmIn) \end{aligned}$$

B. Linear Layer Estimation

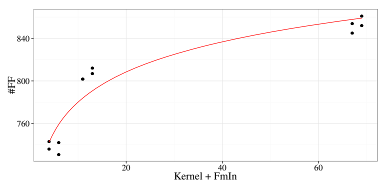
As for the convolutional layers, we synthesized multiple configuration for the linear layers using a number of input neurons ranging from 36 to 4608 and a number of neurons generating outputs from 2 to 64. The models we identified are reported here, and Figure 3 reports a graphical representation of the estimations. The only note here is that FFs could not be estimated using a function of a single variable and we estimated it using two variables instead.



(a) BRAM.



(b) LUT.



(c) FF.

Fig. 2: Experimental data for convolutional layers (black points) and corresponding resource estimation model (red line).

$$\begin{aligned}
F_{Linear,BRAM} &= 0.477541 + \\
&+ 0.003478 * Inputs * Outputs \\
F_{Linear,DSP} &= 5 \\
F_{Linear,LUT} &= 768.232 + \\
&+ 5.858 * log(Inputs * Outputs) \\
F_{Conv,FF} &= 562.22599 + \\
&0.00261 * Inputs - 1.33523 * Outputs
\end{aligned}$$

C. *Classification and Prediction Layer Estimation*

As before we run multiple synthesis varying the number of classes used by the classification layer ranging from 2 to 64. What we obtained is that all the resources present a very low variance across all the synthesis. We then decided to estimate all the resources of the classification layer using the average value obtained during the synthesis. We do not report the plots here for sake of space.

VI. Experimental Evaluation

This section aims at providing an evaluation of the resource estimation models identified in the previous chapter. In order to do this we synthesized 15 networks characterized with different parameters:

- the input image size varies from 16×16 to a 64×64;
- the number of convolutional layers is either 1 or 2, the input and output feature maps varies from 1 to 36, and the kernel dimension is either 3 or 5;
- the number of linear layers is either 1 or 2, with an inputs number ranging from 16 to 1536, and a number of neurons in the range 10 to 210;
- the classification layer always uses 10 classes.

None of the configurations for the layers we used in this evaluation phase have been used to train the models described in the previous section.

Resource estimation has been carried out by applying the models described in the previous section to each of the layers composing the networks, and then summing up the prediction for each type of resource. Table II reports the results obtained. As we can see from the tables, LUTs and DSPs are predicted very good. LUTs have an average relative prediction error of 3.71% with a low variance, and DSPs are wrong predicted by 2 units in all the tests. The result on the DSPs prediction suggests us that a predictive model in this case might not be so useful giving the fact that all the configurations we tested basically use the same amount of this type of resource. FFs are predicted with an average relative error of 15% with a low variance. Given the large amount of FFs nowadays available on FPGAs, we assume that this prediction error, which is on average of 1264 units, is a good result. Finally the sole resource for which we have a bad prediction results is the BRAM. The average relative error is 24% with a very high standard deviation. We suppose that this error might be caused by optimization carried out by the HLS tool to allow faster access to data when multiple layers access shared variables. Nonetheless, if we look at Figure 4 we see that more than 80% of the tested networks has a prediction error of 30 BRAM, which can be considered still a good result.

TABLE II: Summary of the resource estimation results. Table reports for each resource the average and standard deviation of absolute and relative errors across all the tested networks.

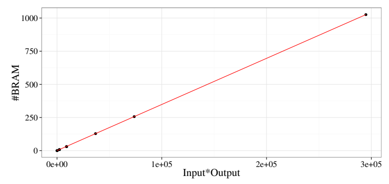
Resource	Avg. Abs. Error	Std. Abs. Error	Avg. Rel. Error [%]	Std. Rel. Error [%]
BRAM	15.82	13.38	24.41	22.53
LUT	430.28	236.09	3.71	1.83
FF	1263.88	182.63	14.88	1.47
DSP	2	0	1.96	0.07

VII. Conclusions and Future Work

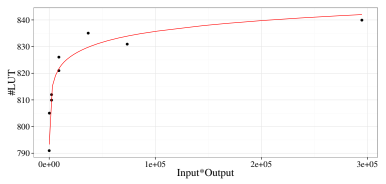
In this paper we presented our web-based framework for the design of CNNs. The framework allows for a faster customization of the network and its realization on a reconfigurable device by directly interfacing with Vivado HLS. The specific focus of this work is the identification of models to predict resource estimation of the CNN under development in order to allow the designer to instantiate more than one network, if possible, to parallelize the computation over the dataset. The models we identified allow to perform a prediction of the resource usage without having to wait the synthesis of the circuit speeding up the development process. As illustrated in the paper the resource estimation is quite accurate except for the BRAM; nonetheless, the absolute error is less than 30 BRAMs in more than 80% of the tested CNN. Future works aims at developing more robust models to further improve prediction accuracy for both BRAM and LUT.

Acknowledgment

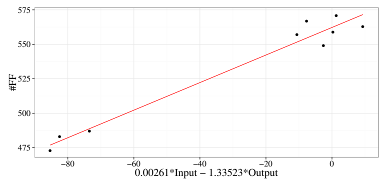
This work was supported by the European Commission in the context of the H2020 FETHPC EXTRA project (#671653).



(a) BRAM.

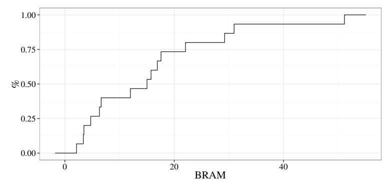


(b) LUT.

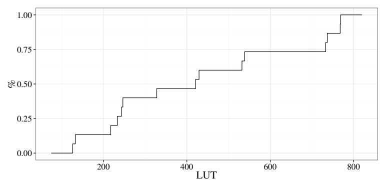


(c) FF.

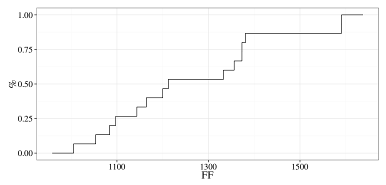
Fig. 3: Experimental data for linear layers (black points) and corresponding resource estimation model (red line).



(a) BRAM.



(b) LUT.



(c) FF.

Fig. 4: Cumulative distributed function of the absolute prediction error.

References

[1] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[2] P. Y. Simard, D. Steinkraus, and J. C. Platt, "Best practices for convolutional neural networks applied to visual document analysis," *2013 12th International Conference on Document Analysis and Recognition*, vol. 2, p. 958, 2003.

[3] S. Ji, W. Xu, M. Yang, and K. Yu, "3D convolutional neural networks for human action recognition," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 35, no. 1, pp. 221–231, 2013.

[4] Y. LeCun, K. Kavukcuoglu, and C. Farabet, "Convolutional networks and applications in vision," in *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, 2010, pp. 253–256.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc., 2012, pp. 1097–1105.

[6] M. Peemen, A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for convolutional neural networks," in *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, Oct 2013, pp. 13–19.

[7] 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 *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on*. IEEE, 2009, pp. 53–60.

[8] 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*. ACM, 2015, pp. 161–170.

[9] E. Del Sozzo, A. Solazzo, A. Miele, and M. D. Santambrogio, "On the Automation of High Level Synthesis of Convolutional Neural Networks," in *Parallel and Distributed Processing Symposium (IPDPS), Reconfigurable Architectures Workshop (RAW), 2016 IEEE International*. IEEE, 2016.

[10] Xilinx Inc., "Vivado HLS." [Online]. Available: http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[11] ——, "Vivado Design Suite." [Online]. Available: http://www.xilinx.com/products/design-tools/vivado.html

[12] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proceedings of the 25th International Conference on Machine Learning*, ser. ICML '08. ACM, 2008, pp. 160–167.

[13] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, "Cnp: An fpga-based processor for convolutional networks," in *2009 International Conference on Field Programmable Logic and Applications*, 2009, pp. 32–37.

[14] M. Peemen, B. Mesman, and H. Corporaal, "Inter-tile reuse optimization applied to bandwidth constrained embedded accelerators," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, ser. DATE '15. EDA Consortium, 2015, pp. 169–174.

[15] C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun, and E. Culurciello, "Hardware accelerated convolutional neural networks for synthetic vision systems," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, 2010, pp. 257–260.

[16] D. Strigl, K. Kofler, and S. Podlipnig, "Performance and scalability of GPU-based convolutional neural networks," in *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, 2010, pp. 317–324.

[17] D. C. Ciresan, U. Meier, J. Masci, L. Maria Gambardella, and J. Schmidhuber, "Flexible, high performance convolutional neural networks for image classification," in *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, vol. 22, no. 1, 2011, p. 1237.

[18] NVIDIA, "Accelerate Machine Learning with the cuDNN Deep Neural Network Library." [Online]. Available: http://devblogs.nvidia.com/parallelforall/accelerate-machine-learning-cudnn-deep-neural-network- library/.

[19] K. Samal, "Fpga acceleration of cnn training," 2015.

[20] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[21] "Torch Framework." [Online]. Available: http://torch.ch