2021 34th International Conference on VLSI Design and 2021 20th International Conference on Embedded Systems (VLSID) | 978-1-6654-4087-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/VLSID51830.2021.00060

An Automated Tool for Implementing Deep Neural Networks on FPGA

Masoud Shahshahani*, Mohammad Sabri†, Bahareh Khabbazan‡, and Dinesh Bhatia*
*Department of Electrical and Computer Engineering, University of Texas at Dallas, Texas, USA
† School of Electrical Engineering, Iran University of Technology, Tehran, Iran
‡ College of Engineering, Tehran University, Tehran, Iran
Email: *(masoud.shahshahani and dinesh)@utdallas.edu,
†Mohammad.sabri@ut.ac.ir, ‡b_khabbazan@alumni.iust.ac.ir

***Abstract*—FPGA based Deep Neural Networks provide the advantage of high performance, highly parallel implementation with very low energy requirements. A designer must consider various configuration choices, processing components, data-flow types, local memory hierarchy, and fixed-data-precision for a DNN implementation. An exploration tool is essential for building a reconfigurable, fast, and efficient DNN hardware accelerator. We present a methodology to automatically create an optimized FPGA-based hardware accelerator given DNNs from standard machine learning frameworks. We generate a High-Level-Synthesis (HLS) code depending on the user preferences with a set of optimization** `pragmas`**. For a faster and cost-effective hardware accelerator, the tool employs a software-model to estimate the execution time and hardware utilization using trained *machine-learning* models. The model evaluation results show that our framework performance speed-up compares well with the state-of-the-art accelerators using Xilinx FPGA platforms.**

***Index Terms*—FPGAs, Accelerators, Deep Neural Networks**

I. Introduction

Deep Neural Networks (DNNs) have demonstrated great success in numerous machine learning applications for prediction and classification. These deep computational models are composed of multiple sequentially arranged processing layers that help to learn the representations within a given data set. These data elements can be processed for various tasks related to object recognition. Convolutional Neural Networks (CNN) are a special class of deep networks that make use of convolution to extract features from (usually a time-domain or frequency-domain) data and then use the extracted features to classify that data for final inferencing.

Most early implementations of DNNs/CNNs made use of GPGPU architectures and have exploited the extreme parallelism offered by modern GPUs. GPUs provide easy access to floating-point operations and also allow very high memory bandwidths. These characteristics make GPUs an ideal candidate for DNN based accelerators. However, keeping *size, weight,* and *power* requirements into consideration, it is the power requirements that make GPUs an unfavorable candidate for DNN implementations. Excessive power dissipation also requires unique thermal management mechanisms in the form of heat sinks and other active/passive cooling methods. These special arrangements can add to the weight limits for the deployment of the end application.

In a recent survey [1], the authors made a comparison of executing some of the prominent DNN examples on GPUs and field-programmable gate arrays (FPGAs). One of the metrics that they offered for comparison in performance per watt. It is believed that FPGA-based accelerators are ideal platforms for implementing Deep Neural Networks due to their high performance to power ratio.

Designing a DNN accelerator requires careful consideration and trade-off between the available resources, performance requirements. The performance captures both the latency as well as energy requirements for a design. A highly parallel circuit would yield high performance but may not be feasible from energy or resources point of view. In addition, the memory bandwidth bottlenecks may not allow us to exploit available parallelism. Optimizing a design by choosing between resources, available architectural options, varied design implementation options, and memory implementation strategies is non-trivial even for a very experienced designer. Each layer's weights can be stored in the on-chip FPGA memory to reduce the data accesses[2, 3]. For large DNNs, it is infeasible because of large size weights and limited on-chip block memories[2]. The design space is simply too complex[4].

Our methodology addresses these inter-related problems and arrives at an efficient hardware accelerator implementation in terms of computational performance and resource constraints on a target platform. The rest of this paper is organized as follows. Related works are reviewed in the section II. In section III, we describe our proposed DNN hardware accelerator framework, the performance estimator, various architecture choices, and optimization techniques to get the most optimum design. The experimental results of the FPGA implementation are compared with some related prior works in section IV. Section V concludes the paper.

II. Related works

Many automated frameworks and design automation tools have been proposed to accelerate FPGA-based accelerators' design and development. The majority of tools tend to support the design of the inferencing engine derived from the modeling frameworks like PyTorch. For example, Zhang[6] proposed a full automation tool that integrates the design of a deep learning framework flow to hardware implementation quickly with

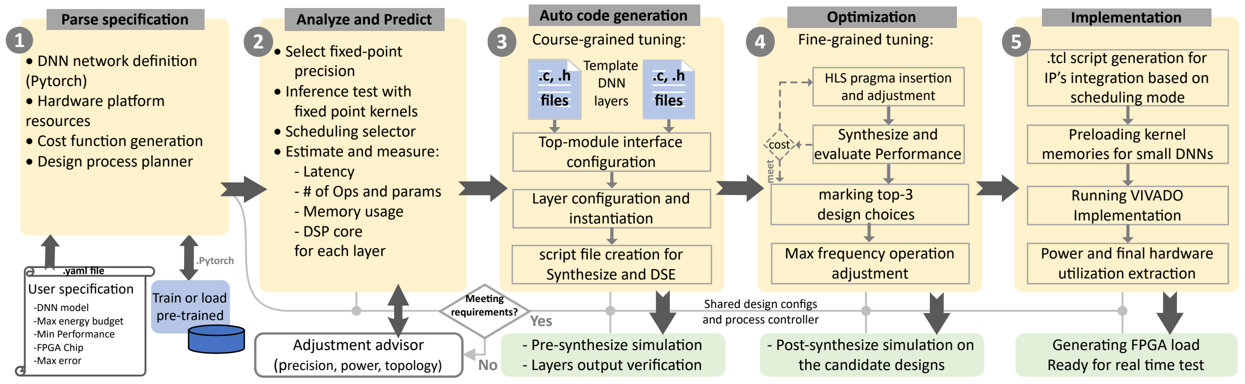
Parse specification
1
• DNN network definition (Pytorch)
• Hardware platform resources
• Cost function generation
• Design process planner
.yaml file
User specification
-DNN model
-Max energy budget
-Min Performance
-FPGA Chip
-Max error
Pytorch
Train or load pre-trained
Analyze and Predict
2
• Select fixed-point precision
• Inference test with fixed point kernels
• Scheduling selector
• Estimate and measure:
- Latency
- # of Ops and params
- Memory usage
- DSP core
for each layer
Meeting requirements?
Yes
No
Adjustment advisor (precision, power, topology)
Auto code generation
3
Course-grained tuning:
.c, .h files
Template DNN layers
.c, .h files
Top-module interface configuration
Layer configuration and instantiation
script file creation for Synthesize and DSE
- Pre-synthesize simulation
- Layers output verification
Optimization
4
Fine-grained tuning:
HLS pragma insertion and adjustment
cost
Synthesize and evaluate Performance
meet
marking top-3 design choices
Max frequency operation adjustment
Shared design configs and process controller
- Post-synthesize simulation on the candidate designs
Implementation
5
.tcl script generation for IP's integration based on scheduling mode
Preloading kernel memories for small DNNs
Running VIVADO Implementation
Power and final hardware utilization extraction
Generating FPGA load Ready for real time test

Fig. 1. Framework to build a DNN hardware accelerator on FPGA [5].

a good acceleration performance. However, it is constrained towards supporting various FPGA architectures. DeepBurning [7] takes a customizable pre-implemented RTL hardware modules that can be configured with the top level defined design parameters. In [8], Jiandong et.al proposed a LoopTree model-based software framework to estimate and model an FPGA-based CNN design accelerator. In Muluken in [9], a DNN accelerator core with and without DSPs is configurable. They lowered the DSP and internal memory utilization by reducing the precision and forcing the synthesizer to use LUTs instead of DSPs for computations. AutoDNNchip [10] work also proposes an automatic DNN accelerator generator that has *Chip Predictor* to predict the design latency and resources with a mathematical model. The prediction model error is smaller than 10% only for a specific architecture platform.

III. DNN Hardware Accelerator Builder

Seeking the highest performance under a limited number of resources plays a vital role in each FPGA-based design. Multiple hardware accelerator can be generated due to the large design space and choices (e.g., memory shape, parallelism factors, loop pipeline depth, data flows, etc.). A framework to build a DNN hardware accelerator is proposed in our previous work[5]. The tool offers a model descriptor that integrates all the design abstraction levels into one shared process controller. Design energy efficiency, throughput, and data precision are all controllable and predictable by tuning the parameters. In this paper, we integrate a new machine-learning model to estimate the accelerator performance before the HLS synthesizes the best architecture process. As shown in Fig.1, the end-to-end tool produces FPGA load from the given specification in five steps: *Parsing user data and training the neural network, analyzing and pre-synthesize estimation, code generation, optimization*, and *implementation*.

An efficient and optimized hardware accelerator is generated in two major phases. First, a coarse-grained tuning quickly configures the FPGA chip hardware architecture, instantiate pre-designed hardware accelerators for each layer, and making the inter-connections. Then, the fine-grained tuning phase explores among intra-optimization parameters of the DNN accelerator to achieve the best possible performance. Various optimization options are also elaborated in III-C.

A. Precision and Performance Estimation

Estimating the design hardware utilization, computation delay for a sample, amount of computation operations (AOs), number of DNN parameters, and data-precision helps the designer configure design parameters to obtain optimized performance. The trade-off between performance and data precision is analyzed in [5]. In this paper, we are proposing a model that estimates the computation latency for each DNN layer.

1) Latency Estimation: The execution time of each DNN layer is equal to the number of cycles needed from loading the layer kernels and input features to compute all the multiplication and addition, and temporarily storing the output features for the next neural network layer. For most designs developed with RTL language, the computation process remains steady for various design scales if there is no interruption in the data path. However, Vivado-HLS scheduling process and the loops iteration interval do not change linearly within various DNN layer scales. Therefore, mathematical estimation is not accurate for most HLS-based designs. Due to the non-linear relation between individual hardware and software parameters, Artificial Neural Networks (ANN) can be used to tackle such problems. There are various techniques in machine learning to solve this problem. In this work, we take the advantage of a MultiLayer Perceptron (MLP) which is composed of a set of nodes that are arranged in layers. Each layer uses the outputs from the previous layers as inputs. Each node is a weighted (W_i)linear combination of inputs (X_i). In this work, input image window size (Im^w), kernels width (K), stride size (S), input (M), and output (N) feature maps are the model inputs. The activation function (f in Eq.1) adds non-linear function to the output neuron and determines whether and to what extent that signal should progress further through the network to affect the outcome. The activation function could

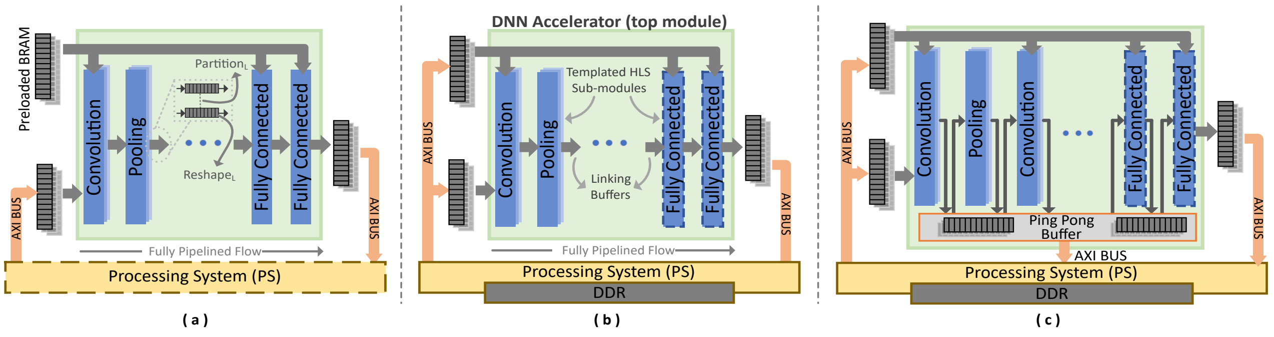

Fig. 2. Three hardware accelerators are available in this framework. Modules with dashed-line are optional in the design to provide more flexibility and trade-off in the accelerator design based on the DNN size and preferences.

be Tanh, Sigmoid, or Relu since the model output has a wide range, Tanh and Sigmoid limit the output scale.

$$Y = f(b + \sum_{i}^{n} X_i \times W_i); \tag{1}$$

In this work, the Convolution and Pooling layers are only modeled for the estimation using a MLP because of three reasons: Firstly, the accelerator for the Fully-Connected (FC) layer has a linear relation with the input/output size which is estimated using Eq.(2),

$$T_{DNN} \approx \sum_{l}^{Layers} \begin{cases} MLP_{model(Im_l^w, S_l, K_l, M_l, N_l)} & l \in CONV, \\ & POOL \\ \dfrac{N_l \times (M_l + 1)}{\beta} & l \in FC \end{cases} \tag{2}$$

Where N_l and M_l here are the number of input and output nodes of each DNN layer l, respectively. β is the reshape factor and the number of parallel multiplication computed in each clock cycle. Secondly, the FC layer computations can be transferred to the soft or hard core processor (PS) as its computation depends on the software code. Thirdly, the FC layers are memory intensive, and their performance depends on the memory bandwidth and latency. Two MLPs are configured to model convolutional and pooling layers and train the network using the prepared dataset. The tool uses a cross-validation technique and shuffles the batches to train the model well. It is important to ensure that the first estimators are formed using a batch of training datasets separate from the one used to form the new dataset. The tool investigates many different configurations (such as learning rate, batch size, number of ANN nodes, and pruning size) to find the best-fitted model for the smallest prediction error. A designer can get an impression of the effects of changes on hardware in moments due to the software predictor model generated and attached to the user HLS design without running an HLS synthesis. The pre-trained model can be re-trained after synthesizing a new design configuration to increase the model estimation resolution.

To evaluate the estimation resolution, we use Eq.(3) to compute the Root Square Mean Error (RMSE) which shows the deviation of the predictions from the mean[11].

$$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2} \tag{3}$$

where y_i are the observed values, $\hat{y}_i$ are the predicted values, and $\bar{y}$ is the mean observed value. Also the goodness of fit was determined by using the cross-validated coefficient of determination (R^2) which can be computed using Eq.(4). R^2 shows how much of the variance of the original dataset is explained by the model and not contributing to the error.

$$R^2 = 1 - \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{\sum_{i=1}^{n} (y_i - \bar{y}_i)^2} \tag{4}$$

2) Precision Modeling: Software-based DNN models are trained using 32-bit floating-point data in CPU/GPU processors, while in the testing or classification phase, high precision is not necessarily required[12]. Reducing data precision from 32-bit floating points to 16 bit and smaller fixed-points of the data reduces the memory requirement as well as the energy for memory transfers. The tool first uses arbitrary quantization to convert floating-point kernels to fixed points, and then execute an inference accuracy analysis to ensure that the output is meeting user preferences. The inference accuracy analyses compare both the pre-synthesis inference simulation in Vivado-HLS and the fixed-point python model to guarantee the auto-generated C++ code's correctness. There is a trade-off in accuracy, power, and resource requirement with different data width. An experimental results for the trade-off of data width versus performance are shown in the IV section.

B. Building the Framework

The tool estimates the total number of DNN parameters (weights), inter-layer data size, DNN inference latency, amount of operations (AOs), and the peak-performance based on the given configuration and model parameters. This information assists the code generator to configure and set an appropriate hardware accelerator before running High-Level synthesis. The kernels and intermediate data of large DNNs cannot be stored in the FPGA memories due to internal memory limitation. So memory and computation resource sharing techniques must be considered. The proposed framework uses the below two steps to build a hardware accelerator engine based on the user's given FPGA platform. Note that this tool supports all type of Xilinx FPGA chips.

1) Hardware Configuration Generation: The computation within the DNN accelerator can be carried out using a pipeline or dataflow architecture. Fig.2 shows three types of proposed accelerators in this framework. The intermediate layers output (layers feature maps) need to be saved to internal (Fig.2(a) and (b)) or external (Fig.2(c)) memories for each layer. Selection of the architecture has great impact on the performance as well as the energy associated with computation.

A fully pipelined architecture offers a high performance accelerator which requires a large amount of memory to store the intermediate data. More DSP cores are needed to achieve a higher performance. In comparison, the semi-pipelined model needs a high bandwidth memory port to swap intermediate data from/to the external memory. In a pipelined architecture, each DNN layer has a specific configuration due to various layer dimensions. This lets the design space explorer (DSE) search among more memory mapping shapes for better optimization. Input data and kernels load from external memory into the local memories. While one sample computation is in progress, the memory controller can transfer the next batch data from external memory to the BRAMs. This technique can reduce the interruption in the computation engine process caused by external memories' communication speed.

The output intermediate data are stored in an external memory by passing through the ping-pong buffer and finally the Direct Memory Controller (DMA). The Fully-Connected (FC) layer computations can be transferred to the FPGA only to have a full FPGA *Programmable-Logic* based platform to lower data-movement between external memory and FPGA. So based on the user's given preferences, DNN model, and available FPGA chip resources, a couple scheduling mode can be selected.

In the proposed FPGA platform, AXI-Master is considered as the main data transfer interface to transfer data between on-chip memory and the external memory. Furthermore, AXI4-Lite is configured to set the DNN accelerator parameters from the processor

After selecting the right architecture, the pre-designed sub-modules (DNN layers) C++ code need to be instantiated and configured from the pre-designed customizable C++ models. The upper bound of loops in the layers, intermediate data-precision, kernels precision, and HLS IP ports are configured based on the given DNN specifications.

C. Architecture Optimization

This phase primarily exploits the High-Level synthesis code annotation based optimizations. Various Vivado-HLS pragmas inserted to reduce the memory space requirements or shorten the loops' latency. Various loop optimization methods such as `unrolling` and `pipelining`, `memory reshaping and partitioning`, and computation cores `pragmas` are inserted in design to optimize the module's (DNN layers in the computation core) computation delay and the overall throughput.

HLS tool aims to maximize the throughput using the minimum possible hardware resource utilization. Providing more hardware resource (DSPs here) and internal memory port let the hardware scheduler to shorten the pipeline initiation interval between the consecutive loop iterations of the nested loop.

Arrays are usually implemented in an FPGA on-chip memory (BRAM), which has two data ports. For DNNs, the input data, output data, and parametric weights can be stored in separate BRAM blocks. This allows multi independent data channels to transfer data to FPGA, which significantly increases memory bandwidth. By reshaping 1-D arrays into 2-D arrays, multiple kernels can be loaded from memory at each cycle, doubling the throughput. This method enhances computation speed with the cost of more logic, while the BRAM memory utilization is constant. The memory access for loading kernels is in-order, while feature map data are not accessible sequentially. Therefore memory reshaping is not always a good solution since not all the data read from one memory address is usable. More memory ports with independent address port resolve this bottleneck. So the linking memories can be partitioned into multiple smaller buffers with cyclic or block order in various dimensions for multi-dimensional memories. The optimum partition type is the cyclic partition, which suits the continuous memory address. The complete partition will be applied when the partition factor equals its BRAM size and is suitable for the layer internal small buffers.

The use of loop-unrolling and array reshaping is very common in the High-Level synthesis. Loop-unrolling increases the degree of parallelism in the execution. However, this process needs to be performed carefully as the parallelism increases the demand on data delivery from the storage and an imbalance between parallelism and available bandwidth can shift the performance bottlenecks from computation to the communication. A careful selection of level of Array Reshaping, Array Partitioning, and Loop Unrolling can positively impact the performance.

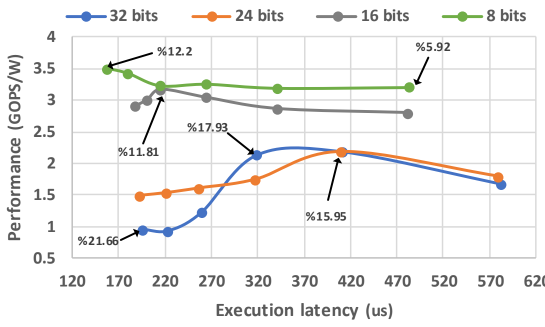

Fig. 3. The diagram shows fine-tuning for 4 different precisions for LeNet. The graph shows the total hardware utilization percentage for the pick-performance, smallest, and largest designs.

We explored key design parameters such as loop optimization techniques and computational resource utilization. Phase 3 and 4 in Fig.1 shows a simulation and verification skeleton used as a state of the art HLS code written with the high-level language. The flow starts from high-level specification and configuration in C++ and uses software programming and debugging tools to ensure that the design is appropriately verified with the written Test-Bench (TB). Once the HLS code functionality is verified, multiple versions of hardware description code are synthesized to explore the trade-off under different performance and area constraints (process on the dashed lines). To confirm the correctness of the final generated RTL for the best combination, we use the automatic co-simulation and formal bit-accurate equivalence checking provided by Vitis-HLS.

As can be seen in Fig.3, some designs come in "large area/fast run-time" while some come in very "small area/slow run-time". All of the HLS optimization pragmas are fixed in this exploration, and only the factor value of array reshaping/partitioning is changed based on the precision to make a constant memory port data width. The best-tuned parameters can be marked as the highest performance within this DSE model. Note that the precision adjustment is not a parameter in the fine-tuning phase. Four different precision are considered for this exploration separately but placed in one graph to have a better analysis in selecting the data precision. The high flexibility of design allows us to adjust general parameters and fit the design subject to the resource allocation and utilization guidelines in the proposed framework.

D. Implementation

Our end-to-end framework generates scripts to execute logic synthesis, placement, and routing, generate FPGA bit-stream, extract estimated power consumption, and area-overhead breakdown. Moreover, our tool preload BRAM memories based on the selected accelerator architecture with respect to the reshape-factor set in the fine-tuning phase. The operating frequency of the computation engine can be tuned at the logic-synthesis phase to increase the computation performance while meeting the FPGA timing constraints.

IV. Experimental Results

The best tuned HLS pragma for each data precision is achieved at the proposed framework's DSE phase. The tool takes 100 samples (among 375,440 input combinations) randomly for training the performance estimator using the range of each hardware accelerator shown in Table I. For all samples, the best tuned HLS pragma, achieved at the DSE phase of the proposed framework, are considered, and the FPGA clock and data-precision are set at 180 MHz and 16-bit fixed-point, respectively. After running a few hyper-parameter tuning steps on the proposed MLP model, we came up with a three-layer neural network that the size of each layer is as follows: 35, 15, 10. ReLU activation function is used to add more non-linearity function to the estimation model layers.

TABLE I
Multi layer Convolution input range as a benchmark. Mean and standard deviation are computed among 100 random sample selection.

		Range	Mean	Std
MLP Input	**Win**	[20 : 200]	110.2	55.2
	LyrIn	[2 : 40]	23.5	9.7
	LyrOut	[2 : 40]	21.7	10.2
	W Kernel	[3 : 7]	5.2	1.6
	Stride	[1 : 3]	1.9	1.1

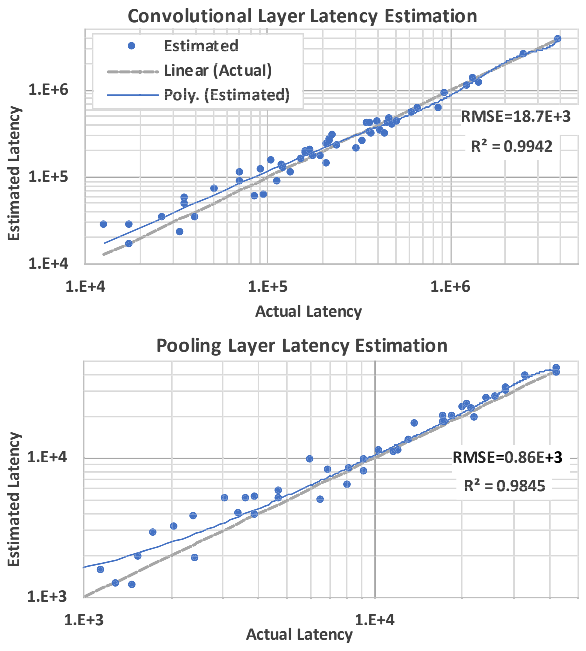

Fig. 4. The Latency estimation of Convolutional and Pooling layers vs the actual values.

The estimated latency of the convolutional and pooling template HLS modules are shown in Fig.4. Estimations are computed based on the given layer inputs in Table I. The predictions for 40 test DNN models were made in less than 200 millisecond. The estimator shows a higher error ratio for smaller designs due to a vast design configuration dimension. This observation also reported in the other research [11]. Having more data to train the network reduces the estimation error.

In this work, the best-tuned framework has been implemented by using Vitis 2020.1, on a Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit FPGA, and running them at the best frequency achieved in DSE. To perform the practical tests, the implemented hardware framework has been integrated with a soft-core processor (ZYNQ) coupled with AXI-Interconnects, and a DMA to handle the communication from/to the external memory. For this experiment, type(a) and type(c) hardware architecture are selected for LeNet and AlexNet DNNs, respectively.

We compare three terms of overall performance, overall area, and the implementation methodology in Table II. Since

TABLE II
PERFORMANCE, POWER EFFICIENCY, PRECISION, AND DESIGN ENTRY COMPARISON WITH PREVIOUS WORKS.

Paper	Yufei [13]	Suda[4]	Stylianos[14]	Bahareh [15]	Alireza [16]	This Work	Muluken [9]	Stylianos[17]	This Work
Automation	No	No	Yes	No	Yes	Yes	No	Yes	Yes
Device	Stratix-V GX7	Stratix-V GX	XC7Z045	XC7Z020	Arria 10	ZCU102	XC7Z045	XC7Z020	ZCU102
Frequency (MHz)	100	120	125	160	199	150	100	100	180
CNN Model	AlexNet	AlexNet	AlexNet	AlexNet	AlexNet	AlexNet	LeNet	LeNet	LeNet
Design Entry	RTL	OpenCL	HLS	RTL	OpenCL	HLS	HLS	HLS	HLS
Precision	16-bit fixed	16-bit fixed	16-bit fixed	8-bit fixed	8-bit fixed	16-bit fixed	16-bit fixed		16-bit fixed
BRAM (KB)	5890	2750	2400	1495	NA	2125	28.5	42	215
DSP	256	220	900	134	300	796	6	4	38
Run Time (ms)	12.75	42.77	8.22	36.53	18.24	11.82 (10.27)[1]	N/A	1.67	0.81 (0.87)[1]
Peak Performance (GOP/S)	114.5	31.8	161.98	40.96[2]	80.04	119.63	0.10	0.48	1.02
Power Efficiency (GOP/S/W)	6.9	N/A	N/A	23.1	NA	11.7	2.5	N/A	3.2

[1] Estimated Run Time based on the proposed model.
[2] Includes only the convolutional layers.

different works exploit different implementation tools and use different FPGA platforms, it is hard to have a straightforward comparison. We have listed different design approaches to have a fair comparison in which almost the same board and the same tool flow are used. It turns out that our design can achieve better performances (same design entry) compared to hardware utilization. We are also aware of the performance gap between our design and the work in [14] while our hardware accelerator uses fewer hardware resources, which reduces the power consumption. So the power efficiency could be similar.

V. CONCLUSION

Deep Neural Networks are an example that will significantly benefit from FPGA based implementation. This paper proposes a methodology to study the relationships between many important design parameters.This understanding helps to arrive at the most optimal design implementation. We have presented various DNN accelerators design using high-level synthesis (HLS) in Xilinx Vitis HLS. We demonstrated the implementation results by optimizing various techniques such as memory reshaping and partitioning, loop unrolling, pipelining, ordering, and more when mapped on the Xilinx Zynq UltraScale+ MPSoC ZCU102 FPGA platform.

REFERENCES

[1] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, O. G. Hock, Y. 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.

[2] M. Shahshahani, P. Goswami, and D. Bhatia, "Memory optimization techniques for fpga based cnn implementations," in *IEEE DCAS*, Dallas, TX, USA, 2018.

[3] C. Zhang and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *FPGA*, Monterey, CA, USA, 2015.

[4] N. Suda and Y. Cao, "Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks," in *FPGA*, Monterey, CA, USA, 2016.

[5] M. Shahshahani and D. Bhatia, "A framework for modeling, optimizing, and implementing dnns on fpga using hls," in *IEEE DCAS*, Dallas, TX, USA, 2020.

[6] X. Zhang and D. Wang, "Dnnbuilder: An automated tool for building high-performance dnn hardware accelerators for fpgas," in *ICCAD*, San Diego, CA, USA, 2018.

[7] Y. Wang, J. Xu, Y. Han, H. Li, and X. Li, "Deepburning: Automatic generation of fpga-based learning accelerators for the neural network," in *DAC*, Austin, Texas, 2016.

[8] J. Mu, W. Zhang, H. Liang, and S. Sinha, "A collaborative framework for fpga-based cnn design modeling and optimization," in *FPL*, Dublin, Ireland, 2018.

[9] M. T. Hailesellasie and S. R. Hasan, "Mulnet: A flexible cnn processor with higher resource utilization efficiency for constrained devices," *IEEE Access*, vol. 7, 2019.

[10] P. Xu and Y. Lin, "Autodnnchip: An automated dnn chip predictor and builder for both fpgas and asics," in *FPGA*, Seaside, CA, USA, 2020.

[11] R. Meeuws and K. Bertels, "High level quantitative hardware prediction modeling using statistical methods," in *IC-SAMOS*, Samos, Greece, 2011.

[12] J. Qiu and H. Yang, "Going deeper with embedded fpga platform for convolutional neural network," in *FPGA*, Monterey, CA, USA, 2016.

[13] Y. Ma, N. Suda, Y. Cao, and J. sun Seo, "Alamo: Fpga acceleration of deep learning algorithms with a modularized rtl compiler," *The VLSI Journal*, 1 2018.

[14] S. I. Venieris and C. Bouganis, "Latency-driven design for fpga-based convolutional neural networks," in *FPL*, Ghent, Belgium, 2017.

[15] B. Khabbazan and S. Mirzakuchaki, "Design and implementation of a low-power, embedded cnn accelerator on a low-end fpga," in *DSD*, Chalkidiki , Greece, 2019.

[16] A. Ghaffari and Y. Savaria, "Cnn2gate: Toward designing a general framework for implementation of convolutional neural networks on fpga," 2020.

[17] S. Venieris and C.Bouganis, "fpgaconvnet: Mapping regular and irregular convolutional neural networks on fpgas," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 30, no. 2, 2019.