2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines

Customizing Neural Networks for Efficient FPGA Implementation

Mohammad Samragh, Mohammad Ghasemzadeh, and Farinaz Koushanfar
ECE Department, UC San Diego , La Jolla, CA 92037, USA
{*msamragh, mghasemz, fkoushanfar*}*@ucsd.edu*

***Abstract*—We propose a novel end-to-end framework to customize execution of deep neural networks on FPGA platforms. Our framework employs a reconfigurable clustering approach that encodes the parameters of deep neural networks in accordance with the application's accuracy requirement and the underlying platform constraints. The throughput of FPGA-based realizations of neural networks is often bounded by the memory access bandwidth. The use of encoded parameters reduces both the required memory bandwidth and the computational complexity of neural networks, increasing the effective throughput. Our framework enables systematic customization of encoded deep neural networks for different FPGA platforms. Proof-of-concept evaluations on four different applications demonstrate up to 9-fold reduction in memory footprint and 15-fold improvement in the operational throughput while the drop in accuracy remains below 0.1%.**

***Keywords*-Deep neural networks, Reconfigurable computing, Domain-customized computing, Platform-aware customization.**

I. Introduction

Deep neural networks (DNNs) have been successfully adopted in many applications, including but not limited to computer vision [1], voice recognition [2], [3], natural language processing [4], and health care [5]. A DNN is composed of multiple layers of neurons stacked in a hierarchical formation. DNN computations involve parameters whose values are learned in a training procedure. Once the parameters are trained, The DNN can extract meaningful features from raw inputs.

DNN computation can be parallelized across multiple computational units, motivating prior work on the efficient implementation of neural networks on GPU, CPU clusters, and FPGA clusters [6], [7], [8]. Many applications require DNNs to be deployed on mobile and embedded devices where memory and runtime budgets are strictly limited. For instance, object recognition in robotics and self-driving cars should be performed in real-time under certain resource constraints[9]. Since FPGAs are promising platforms for high-throughput and power efficient implementation of parallel algorithms, efficient FPGA realization of DNNs on single FPGA chips has attracted researchers' attention [10], [11], [12]. These works are mainly focused on efficient implementation of existing models rather than modifying the models to tailor them for FPGA platforms.

Neural networks are inherently approximate models, and can often be simplified. Previous work shows that there is a significant redundancy among neural network parameters [13], offering alternative models that can deliver the same of accuracy with less computation or storage requirement. Inspired by this fact, we propose a novel end-to-end framework to customize execution of deep neural networks on FPGA platforms. Our framework employs a reconfigurable clustering approach that encodes the parameters of deep neural networks in accordance with the application's accuracy requirement and platform constraints.

Towards this goal, this paper focuses on two challenges associated with execution of DNNs on FPGA settings: first, the memory requirement of fully connected layers often exceeds the available on-chip storage capacity, making it inevitable to store the parameters in the off-chip memory; consequently, the achievable throughput becomes bounded by the communication bandwidth. To address this issue, we propose a greedy algorithm that encodes DNN parameters in accordance with the platform's memory constraints and the application's accuracy requirement. Second, matrix-vector multiplications often require significant amounts of processing resources allocated to MAC operations. To decrease the computational burden, we propose a factorized version of matrix-vector multiplication in the encoded domain that replaces the majority of the MAC operations with additions, allowing efficient use of the FPGA's computing power.

To implement DNNs on different FPGA platforms, We propose a systematic design flow involving two steps. First, to specify the encoding bit-width, the pre-trained parameters of the DNN are encoded in a software module according to (i) the error tolerance, and (ii) the maximum storage capacity that the hardware designer allocates to model parameters. Second, a performance profiling process utilizes the encoding bit-width and the DNN topology to optimally map the corresponding computations to the target platform. This step is performed by specifying parallelism factors within different DNN layers. Eventually, encoded DNN parameters and customized DNN layers are used in design synthesis.

We evaluate the proposed framework on four DNN applications. In particular, we synthesize baseline DNNs and their corresponding encoded DNNs on a Zynq-ZC706 board, then compare them in terms of memory footprint, throughput, accuracy, and resource utilization.

In summary, the contributions of this paper are as follows:

- Devising a greedy algorithm for encoding the pa-

978-1-5386-4037-1/17 $31.00 © 2017 IEEE
DOI 10.1109/FCCM.2017.43

rameters of a pre-trained DNN in accordance with the application's accuracy requirement. This algorithm leverages a clustering-based approach to finding the optimal representatives for the encoding such that DNN accuracy is retained.

- Developing a software API for systematic customization of the encoded parameters with regards to memory constraints. The proposed methodology iteratively searches for the smallest encoding bit-width that ensures classification accuracy. Proof-of-concept evaluations demonstrate up to 9-fold reduction in memory footprint and 15-fold improvement in operational throughput while accepting less than 0.1% drop in classification accuracy.
- Proposing a systematic design flow to optimally schedule DNN computations according to the platform. We formulate computation efficiency to ease performance optimization of DNNs on various platforms.

II. PRELIMINARIES

This paper proposes a novel feature encoding algorithm to customize DNNs for execution on FPGA platforms. Our encoding algorithm is inspired by [14], where the authors utilize a random hash function to decrease the number of free parameters within DNN models. We propose a clustering-based methodology to encode DNN parameters. Our algorithm enables systematic customization of neural networks in accordance with application requirements and hardware constraints. The rest of this section provides insightful background regarding neural networks and our encoding algorithm.

A. Deep Neural Networks

As indicated in figure 1, a DNN is formed by stacking multiple dense layers. Each layer takes an input vector, extracts abstract features from it, and feeds the extracted feature vector to the next layer. The output of the last layer is then used for classification or regression purposes.

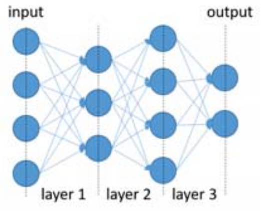


Figure 1: Schematic depiction of a DNN consisting of 3 dense layers. Circles denote vector elements (neurons) and arrows denote matrix elements (parameters).

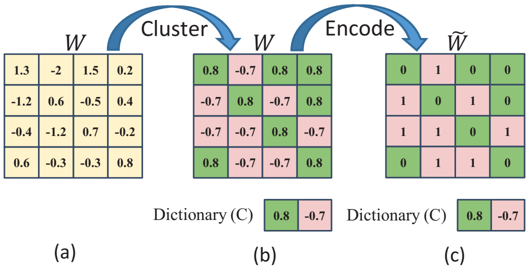
Each DNN layer takes an input vector $X_{N\times 1}$, then computes its output $Y_{M\times 1}$ according to equation 1:

$$Y = f(WX + b) \tag{1}$$

where $f(.)$ is a nonlinear function, $W_{M\times N}$ is a 2-D matrix multiplied by the input, and $b_{M\times 1}$ is a vector of bias values. The matrix W and the vector b are known as the parameters of dense layers. Prior to execution, the parameters are learned in a training process. Once the DNN is trained, it can be executed on the FPGA platform.

B. Parameter Encoding

Figure 2 illustrates the idea of parameter encoding. Given a parameter matrix, $W_{M\times N}$, we seek a dictionary of K values $C = \{c_1, c_2, ..., c_K\}$ to replace the elements of W. Once we find such a dictionary, the original matrix W can be represented by an alternative matrix $\widetilde{W}_{M\times N}$ whose elements are codes referring to dictionary values. The objective of our encoding algorithm is to optimally choose this "dictionary" based on the underlying platform constraints and application accuracy requirements, such that parameter encoding minimally affects DNN functionality. Since the number of values in the dictionary is limited by K, the encoded parameters are represented with $log(K)$ bits, resulting in significant reduction in memory footprint.



1.3	-2	1.5	0.2
-1.2	0.6	-0.5	0.4
-0.4	-1.2	0.7	-0.2
0.6	-0.3	-0.3	0.8

0.8	-0.7	0.8	0.8
-0.7	0.8	-0.7	0.8
-0.7	-0.7	0.8	-0.7
0.8	-0.7	-0.7	0.8

0.8	-0.7

0	1	0	0
1	0	1	0
1	1	0	1
0	1	1	0

0.8	-0.7

Figure 2: Illustration of the encoding method for a 4 × 4 matrix. (a) The original matrix containing floating-point numerals. (b) Matrix elements substituted with a dictionary of 2 values. (c) The encoded matrix of 1-bit numerals alongside the dictionary of floating-point values.

C. Factorized Dot Product

Encoding the parameters not only reduces their memory footprint but also decreases the total FLOPs required to execute DNNs. Consider a dot product involving vector X and a row of the parameter matrix W:

$$dot(W,X) = \sum_{i=1}^{N} W[i] \times X[i] \tag{2}$$

Figure 3 illustrates how dot products can be simplified using encoded parameters: MAC operations involving weights with the same color can be factorized, thus the result can be computed using less computing resources. To facilitate the factorized computation, a factorized coefficient vector V is computed using equation 3.

$$V[i] = \sum_{j\in S_i} X[j] \tag{3}$$

where S_i denotes a subset of W pointing to the i^{th} element of the dictionary:

$$S_i = \{j|W[j] = c_i\} \ or \ equivalently \ S_i = \{j|\widetilde{W}[j] = i\} \tag{4}$$

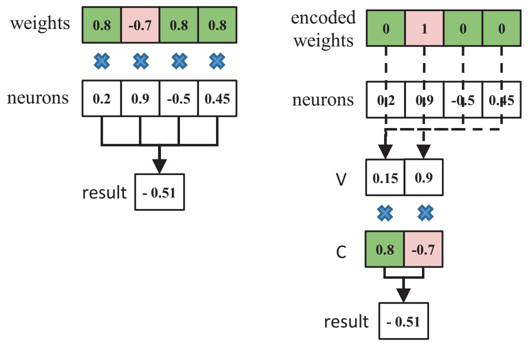


Figure 3: Conventional dot product (left) versus dot product with factorized coefficients (right)

Table I: Memory footprint and computational complexity for three different implementations of dot products.

method	memory (bits)	# add	# mult
no encoding	$32 \times N$	N	N
encoded, not factorized	$log(K) \times N + 32 \times K$	N	N
encoded, factorized	log(K)×$N + 32 \times K$	N+K	K

Vector V has K elements, with K being equal to the number of elements in the dictionary. Equation 5 presents how the correct result is computed by a light-weight dot product over factorized vector V and dictionary C.

$$dot(W,X) = \sum_{i=1}^{N} W[i] \times X[i] = \sum_{k=1}^{K} V[k] \times C[k] \qquad (5)$$

The right-most summation in equation 5 requires only K multiply-accumulate operations. Table I summarizes the memory footprint and computational complexity of a dot product involving N elements for three different approaches.

III. Previous Work

Previous work shows that there is a large amount of redundancy among neural network weights [13]. One observation is that the numerical precision can be customized in accordance with different applications. Many authors have suggested the use of fixed-point parameters of variable bitwidths to customize DNNs for efficient execution [15], [16], [17], [18], [19]. The design process of reconfigurable accelerators with variable numerical precision can be troublesome [20]. Additionally, the use of fixed-point parameters is application-specific and does not provide a generic solution for applications which require higher numerical precision [15].

Instead of using fixed-point parameters, this paper assumes high-precision arithmetics, i.e. floating point operations, to be applicable to generic applications. Our encoding approach is inspired by parameter hashing [14]. Similar idea has been proposed to design specific neural networks based on look-up tables [21]. The focus of [14] is training DNNs on CPU or GPU, thus they only report the number of free parameters (i.e. the dictionary of cluster centroids) as the memory footprint. Random hashing requires large dictionaries to ensure DNN accuracy hence the encoded parameters incur high memory overhead, resulting in inefficient FPGA designs in practice. Our clustering-based approach retains the desired classification accuracy with significantly smaller dictionaries (K=8 in our experiments, 3-bits per encoded weight) since it explicitly leverages the structure of the parameter space. As such, our algorithm is able to customize the memory footprint of the encoded parameters according to the memory constraints of the target platform.

A line of existing work demonstrates that the throughput can be significantly improved by systematic computation mapping methodologies either in single-chip accelerators [10] or clusters of processing units [8]. These works efficiently map DNN computations to FPGA resources, however they lack a model adaptation technique to customize DNNs for the underlying platform. The works in [22], [23] propose platform-aware dimensionality reduction for the input data, while our work is focused on the model itself. Therefore, existing works are orthogonal to this paper. Our methodology reduces the total memory footprint and computational burden of DNNs, which can be applied on top of existing frameworks to further improve their performance.

IV. System Overview

Figure 4 illustrates the design flow of customized-encoded neural networks. First, the “Customized Encoding Module” encodes pre-trained neural networks in accordance with the memory constraints and/or the desired DNN accuracy. Next, the parameterized HLS functions are utilized to form the corresponding DNN topology. These functions are customized for the specifications of the underlying platform within the “Kernel Customization” block. Finally, the customized kernels along with the encoded parameters are used to synthesize the DNN. The rest of this section provides details of the framework.

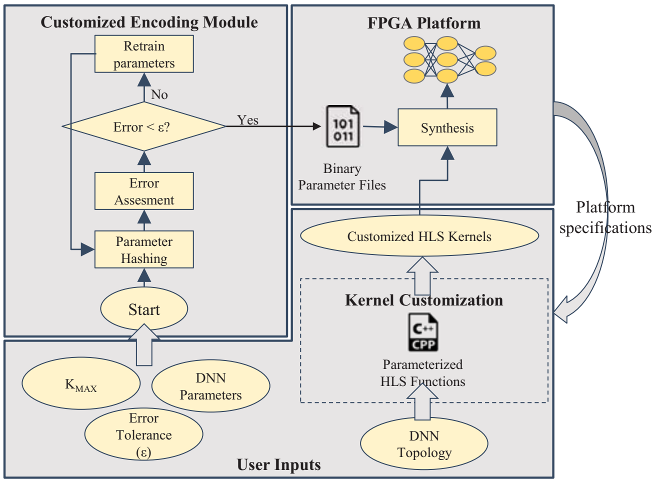


Figure 4: Global design flow of encoded neural networks.

A. Customized Encoding Module

This module takes a pre-trained DNN and encodes its parameters using our proposed greedy algorithm that clusters the parameters considering the amount of memory that the designer wishes to allocate to DNN parameters. In particular, the software module customizes the encoding bit-width to ensure a certain level of accuracy.

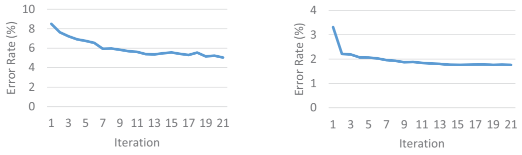
1) Encoding Algorithm: We propose a greedy algorithm to encode DNN parameters with minimal loss of accuracy. As illustrated in figure 4, the algorithm has 3 steps: weight clustering, error assessment, and weight retraining. Below we describe each step in details.

Weight clustering: The weight clustering module applies the K-means clustering algorithm to the original matrix of parameters, with K being equal to the number of elements in the dictionary. Given a matrix W, the objective of K-means is to minimize the numerical distance between clustered values and the original values. The optimization objective is presented in equation 6.

$$\min_{c_1,\ldots,c_K} \Big(\sum_{k=1}^{K} \sum_{W_{ij}\in c_k} ||W_{ij} - c_k||^2\Big) \tag{6}$$

Error estimation: Clustering affects the numerical computations within the DNN model, which may result in an increased error rate. The error assessment module calculates the classification error over the cross-validation data and terminates the algorithm if the error rate is within the threshold provided by the user. The error rate can be controlled in two ways: (i) increasing the number of dictionary elements, and (ii) clustering and retraining the parameters iteratively.

Weight retraining: The retraining module fine-tunes clustered parameters for a pre-specified number of training epochs. A retrained DNN model is more robust against weight clustering since its trained parameters were initialized with clustered values; therefore, iterative retraining/clustering reduces the error of the model. Figure 5-a shows an example of DNN error monitored in subsequent cluster/retrain iterations for $K = 2$. Note that the improvement in the accuracy shows diminishing returns in subsequent iterations. Figure 5-b shows the monitored error for the same DNN with $K = 4$, showing that the asymptotic error rate can be decreased with larger K.



(a) K=2

(b) K=4

Figure 5: Error rate of encoded DNN for K=2 and K=4.

2) Software API: We provide a software API to encode pre-trained DNN parameters according to the training data, the maximum dictionary size, and the threshold for validation error. It runs the encoding algorithm with $K \in \{2 \ldots K_{max}\}$ to find the minimum K resulting in error rate below the threshold ε, where K_{max} and ε are user-defined parameters. The encoded parameters are stored in binary files which will be used for initialization of DNN parameters during synthesis.

B. FPGA Execution Module

The binary files containing encoded/raw parameters and the available HLS functions are used to synthesize DNNs for different applications. We have developed basic building blocks of DNNs which allow the designers to implement DNNs of arbitrary topologies. Each building block is a parametrized function implemented in high-level synthesis (HLS) programming language.

1) Matrix-vector Multiplication Kernel: The most computation-intensive operation in DNN execution is matrix-vector multiplication. Figure 6 presents our approach to parallelize such an operation. Two levels of parallelism controlled with parameters P_1 and P_2 are leveraged in each dense layer. First, multiple rows of the parameter matrix W are processed simultaneously in the "DNN Layer" block: each row is accompanied by a copy of the input vector and the pairs are fed into "dot product" modules. Second, each "dot product" function partitions its input vectors and concurrently executes MAC operations over the partitioned subsets. The accumulated results of the subsets are added together within the "reduce sum" block to compute the output. As mentioned previously, our HLS functions are parameterized to ease systematic design. Parameters P_1 and P_2 along with input/output dimensionality are passed to HLS functions, enabling designers to optimally customize the kernels for execution on different FPGA platforms. We will elaborate on kernel customization in section IV-C.

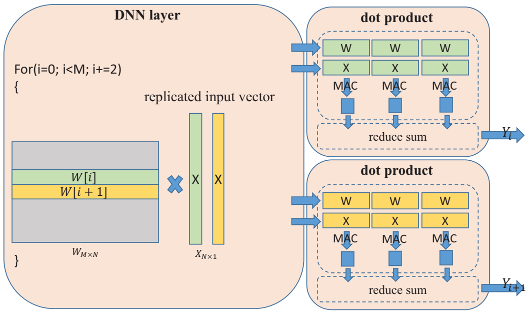


Figure 6: An example of parallelized matrix-vector multiplication with $P_1 = 2$ and $P_2 = 3$.

2) Encoded Matrix-vector Multiplication Kernel: This module is similar to the conventional matrix-vector multiplier, except that one of the MAC operands should be decoded prior to multiplication. Figure 7 illustrates the difference between multiplication of raw inputs and encoded inputs. The decoder is realized by a small look-up table which imposes a negligible overhead. In the case of storing parameters in off-chip memory, the throughput is upper-bounded by the communication bandwidth:

$$Throughput\ upper\ bound = \frac{communication\ bandwidth}{No.\ bits\ per\ FLOP}$$

As such, while an operation involving a floating point parameter requires 32 bits to be read from the off-chip memory for each element of the matrix, an operation on an encoded parameter requires fewer bits (2-3 in our experiments) to be accessed; therefore, for the same communication bandwidth, the encoded DNN module is able to load more parameters from the memory in a fixed amount of time. Additionally, the encoded weights may even fit inside the on-chip BRAMs of the FPGA, in which case the throughput is not limited by the communication bandwidth.

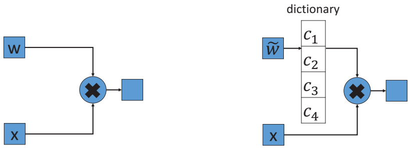


Figure 7: Multiplication with raw weights (left) and encoded weights (right).

3) Factorized Matrix-vector Multiplication Kernel: As previously discussed in section II-C, the dot products operating on encoded inputs can be performed in a factorized way to reduce the computational burden. First, equation 4 is used to compute the vector of factorized coefficients V, then the light-weight dot product over vector V is computed according to equation 5. The factorized coefficients are computed in one pass over input vector X as in pseudo code 1.

Algorithm 1 Pseudo code for computing factorized coefficients.

```
inputs: $\widetilde{W}$, $X$
outputs: $V$
$V \leftarrow 0$
for $i = 1 \dots N$ do
    $V[\widetilde{W}[i]] \leftarrow V[\widetilde{W}[i]] + X[i]$
end for
return $V$
```

Figure 8 demonstrates the flow diagram for a factorized dot product. Similar to the non-factorized DNN layer, two levels of parallelism are leveraged in the factorized DNN layers: parameter P_1 determines the number of concurrent dot products and parameter P_2 sets the parallelization factor along the "factorize" kernel. The "factorize" block involves only addition operations, hence it requires fewer resources than conventional dot products. The "reduce sum" block within the "factorize" module computes the sum over partially factorized vectors and stores the result in vector V, which is then used by the light-weight dot block to compute the output in few cycles.

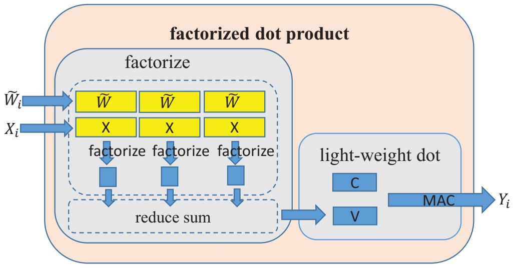


Figure 8: Factorized dot product kernel with parallelization factor $P_2 = 3$.

C. Kernel Customization

Let $\{(P_1^1, P_2^1), \dots, (P_1^L, P_2^L)\}$ be the parallelization factors applied in a DNN composed of L layers, where superscripts denote layer ID and subscripts denote the dimension along which the parallelization is applied. Figure 9 presents our methodology to find optimal parallelization factors, which in turn determines how computations are mapped to FPGA resources. Below we describe the 3 steps of the kernel customization process in details.

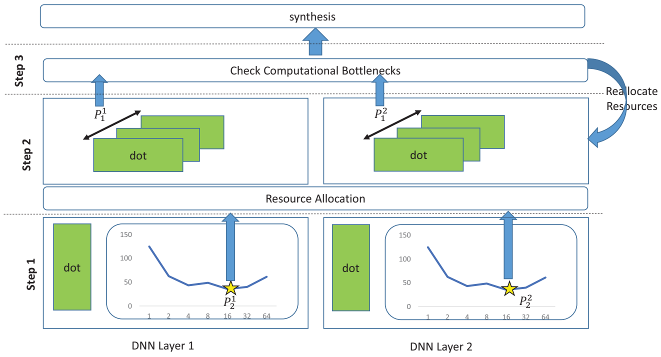


Figure 9: The 3-step methodology for kernel customization.

Step 1, assigning P_2 to each layer: For each DNN layer, we consider the "dot product" kernel and profile synthesis reports corresponding to different parallelization factors. Table II outlines metrics used for optimizing the kernel. We define the effective utilization (**EU**) metric so that resource utilization and computation time are jointly involved in the optimization process. A smaller **EU** corresponds to higher throughput achieved with less resource utilization; as such, the optimization objective is to minimize **EU** by exploring possible parallelization factors.

Figure 10 presents design metrics for three different realizations of an example dot function, where dot_{base} executes conventional dot product with raw weights, dot_E uses encoded weights without factorization, and dot_{EF} computes its output using encoded weights and factorized coefficients. The following observations are worth noting:

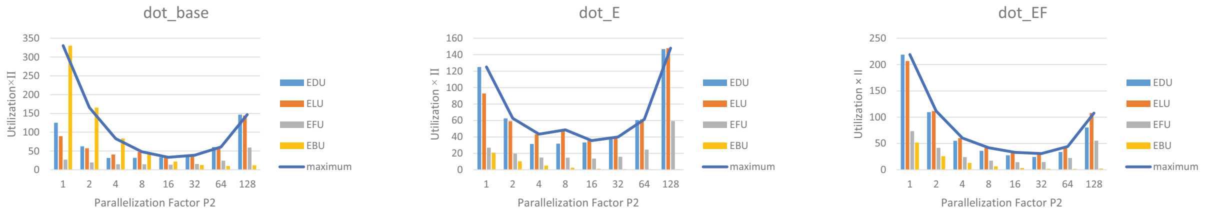


Figure 10: Synthesis report of dot product kernels processing vectors of length $N = 5625$. The minimum of each curve corresponds to the optimal P_2. Metrics used in this figure are defined in table II.

Table II: Metrics defined to assign parallelizarion factor P_2.

metric	discription
Initiation Interval (**II**)	No. cycles to compute the function
Effective DSP Utilization (**EDU**)	DSP Utilization (percent) × **II**
Effective LUT Utilization (**ELU**)	LUT Utilization (percent) × **II**
Effective FF Utilization (**EFU**)	FF Utilization (percent) × **II**
Effective BRAM Utilization (**EBU**)	BRAM Utilization (percent) × **II**
Effective Utilization (**EU**)	Max (**EDU, ELU, EFU, EBU**)

- The minimums of the curves correspond to the optimal P_2 for the kernels.
- The baseline function (dot_{base}) has high BRAM utilization, imposing high **EU** and restricting the throughput.
- The **EDU** and **ELU** of dot_{EF} are smaller than those of the other two kernels, making it suitable for FPGAs with lower computing power.

Step 2, assigning P_1 to each layer: The next step is to determine how many “dot product” kernels should be instantiated in each layer. To do so, we first allocate a portion of the available resources to each DNN layer based on its computational complexity, which is proportional to the number of parameters within the layer. For each layer, parameter P_1 is determined such that the maximum number of “dot product” kernels are instantiated without violating the assigned resource budget.

Step 3, avoiding computational bottlenecks: The operational throughput of DNNs can be improved by pipelining the execution of DNN layers. To achieve maximum throughput, it is important to make sure that DNN layers have latencies of the same order. Figure 11 illustrates how high-latency layers can limit the operational throughput. The throughput can be improved by exchanging the resource budget in favor of DNN layers with high latencies, and reassigning parallelization factors as in step 2.

V. Evaluation

Our design experiments are synthesized for the Zynq-ZC706 board using Vivado HLS 2015.4. Table III outlines the specifications of the FPGA and the communication bandwidth. In this section, we evaluate the effect of customized parameter encoding on four different DNN architectures. In particular, we compare baseline DNNs with their corresponding encoded DNNs in terms of accuracy, memory footprint, and throughput.

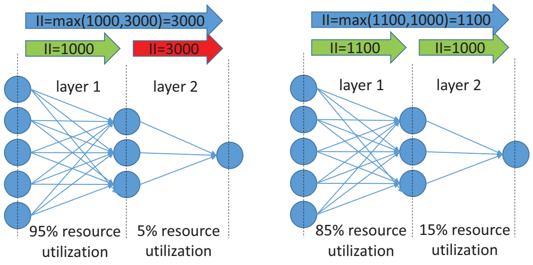


Figure 11: Left: high initiation interval is imposed by layer2. Right: the overall initiation interval is improved by exchanging resources among DNN layers.

Table III: Specifications of the Zynq-ZC706 board.

DSP	LUT	FF	BRAM	PCIE bandwidth
900	218600	437200	1090	up to 20Mbps

A. Datasets

We evaluate the framework on four datasets, each of which has a different data dimensionality and DNN topology outlined in table IV.

Speech Recognition: Many mobile applications require online processing of vocal data. We evaluate the framework on the ISOLET dataset [24]. The goal of this task is to classify vocal signals to one of the 26 English letters.

Indoor Localization: We apply a DNN to the Indoor Localization dataset [25]. Automatic user localization using GPS is widely used in today’s mobile phones; however, the loss of GPS signal in indoor environments inspires the use of machine learning algorithm for the task.

Activity Recognition: For This dataset, the objective is to recognize human activity based on signals collected from motion sensors [26].

MNIST: MNIST is a popular machine learning dataset including images of handwritten digits [27]. The objective is to automatically classify handwritten inputs to one of the ten digits $\{0 \ldots 9\}$.

B. Effect of Parameter Encoding on Accuracy

As discussed previously, the encoding module is used to customize the DNN for different applications and platforms. We use Keras [28] and Scikit-learn [29] libraries for software realizations of DNNs and K-means algorithm respectively.

Table IV: Datasets and DNN topologies

Application	input dim.	categories	DNN Topology
Speech Recognition	617	26	$617 \times 512 \times 512 \times 26$
Indoor Localization	520	13	$520 \times 512 \times 512 \times 13$
Activity Recognition	5625	19	$5625 \times 512 \times 512 \times 19$
MNIST	784	10	$784 \times 512 \times 512 \times 10$

Table V: Classification accuracy for different DNN applications. Results in bold letters correspond to the encoding bit-widths chosen by the customized encoding module.

dataset	baseline	K=2	K=4	K=8
Speech Recognition	96	89.5	95.1	**96**
Indoor Localization	95.2	93.2	95.1	**95.2**
Activity Recognition	98.4	89	**98.3**	98.4
MNIST	98.4	94.9	98.3	**98.4**

Hidden layers of all DNNs apply “Rectified Linear Unit” on their outputs. For each experiment, the baseline DNN is trained until its test accuracy stops improving. Stochastic gradient descent with momentum [30] is used for the training and customization processes. Dropout [31] is applied during training to avoid over-fitting. We centralize the *Speech Recognition* and *Activity Recognition* datasets prior to training, such that input features have zero mean and unity standard deviations, which results in slight improvement in baseline accuracies.

Table V presents the accuracy of baseline DNNs and DNNs encoded with different numbers of clusters. We do not report accuracy for more than 8 clusters as they result in the same accuracy for higher dictionary sizes. Results in bold letters correspond to dictionary sizes for which either the entire DNN fits inside the BRAMs or the baseline accuracy is recovered. Our clustering-based algorithm is able to fully retain the baseline accuracy for *Speech Recognition*, *Indoor Localization*, and *MNIST* datasets, while the drop of accuracy for *Activity Recognition* is below 0.1%.

In these examples, the encoded parameters fit in BRAMs. In general, access to off-chip memory might be unavoidable since some large-scale DNNs might not fit in BRAMs even with 1-bit encoding. Also, the desired accuracy might not be achievable using small dictionaries, leading to high memory footprint of the encoded parameters. It is worth mentioning that, even in such scenarios, encoding the parameters improves the operational throughput as it reduces the total bandwidth required to access the parameters.

C. Comparison of Customized DNN Architectures

For each application, we compare synthesis reports of 3 different realizations of the DNN: (i) a baseline neural network that uses 32-bit floating-point numerals denoted as DNN_{base}, (ii) an encoded DNN without factorized multiplication denoted as DNN_E, and (iii) an encoded DNN that uses factorized matrix-vector multiplication denoted as DNN_{EF}. Each DNN is implemented using HLS kernels customized as described in section IV-C. Table VI presents customized parallelization factors for DNN architectures.

Table VI: Customized parallelization factors

Application	Layer 1 (P_1, P_2)	Layer 2 (P_1, P_2)	Layer 3 (P_1, P_2)
Speech Recognition	(24,8)	(24,8)	(1,8)
Indoor Localization	(24,8)	(24,8)	(1,8)
Activity Recognition	(16,32)	(8,8)	(1,8)
MNIST	(24,8)	(24,8)	(1,8)

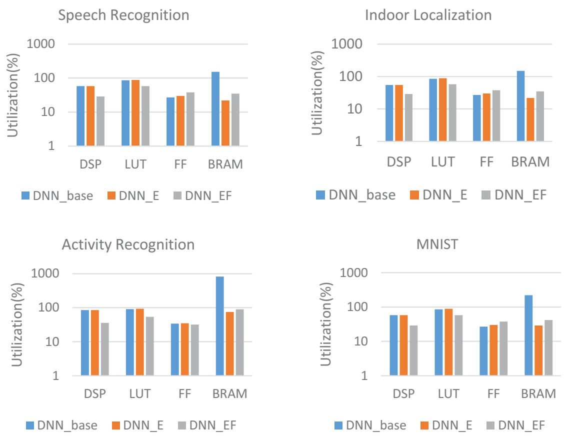


Figure 12: Synthesis reports for DNN architectures. DNN_{base} is the baseline, DNN_E is the encoded neural network without factorized dot kernels, and DNN_{EF} denotes encoded neural network with factorized dot products.

For each application, we adopt the customized encoding bit-width from table V, then use the parallelization factors of table VI along with the parameterized HLS kernels to synthesize the neural network. The final synthesis reports are outlined in figure 12. Each sub-figure compares synthesis reports of the three DNN architectures – namely DNN_{base}, DNN_E, and DNN_{EF} – for one of the datasets. Note that the vertical axes are scaled logarithmically.

In all experiments, the BRAM utilization of DNN_{base} exceeds the maximum capacity, while DNN_E and DNN_{EF} remain below the on-chip memory budget; therefore, the operational throughput of DNN_{base} is bounded by

$$\frac{communication\ bandwidth}{bits\ per\ FLOP} = \frac{20Gbps}{32\ bpFLOP} = 625MFlops$$

as it should read one parameter from the off-chip memory for each operation. Table VII compares the throughput of the three DNNs for a clock frequency of 100 MHz. The encoded DNNs improve the baseline throughput by an order of magnitude. DNN_E and DNN_{EF} can be alternatively used based on FPGA specifications. In our case studies, the operational throughput of DNN_{EF} is on average 11.5% less than that of DNN_E. However, the normalized throughputs of DNN_{EF} with respect to DSP utilization and LUT utilization are on average 1.8× and 1.4× higher than those of DNN_E

Table VII: Synthesis reports of practical design experiments.

	Speech Recognition			**Indoor Localization**			**Activity Recognition**			**MNIST**			**Average (Normalized)**		
	DNN_{base}	DNN_E	DNN_{EF}	DNN_{base}	DNN_E	DNN_{EF}	DNN_{base}	DNN_E	DNN_{EF}	DNN_{base}	DNN_E	DNN_{EF}	DNN_{base}	DNN_E	DNN_{EF}
Nominal Throughput (GFLOPs)	6.57	6.56	6.03	6.89	6.89	5.46	10.1	10.1	9.91	6.12	6.11	5.21	1	0.99	0.88
Operational Throughput (GFLOPs)	0.625	**6.56**	6.03	0.625	**6.89**	5.46	0.625	**10.1**	9.91	0.625	**6.11**	5.2	1	11.86	10.64
Throughput/DSP (GFLOPs/DSP)	1.2E-3	1.2E-2	**2.3E-2**	1.3E-3	1.4E-2	**2.1E-2**	8.2E-4	1.3E-2	**3E-2**	1.2E-3	1.2E-2	**2E-2**	1	11.66	22.14
Throughput/LUT (GFLOPs/LUT)	3.3E-6	3.4E-5	**4.8E-5**	3.3E-6	3.5E-5	**4.3E-5**	3.1E-6	5E-5	**8.4E-5**	3.3E-6	3.1E-5	**4.1E-5**	1	9.51	16.77
Throughput/FF (GFLOPs/FF)	5.3E-6	**5E-5**	3.6E-5	5.3E-6	**5.2E-5**	3.3E-5	4.2E-6	6.6E-5	**7.1E-5**	5.3E-6	**4.7E-5**	3.1E-5	1	8.99	8.95
Throughput/BRAM (GFLOPs/BRAM)	-	**2.7E-2**	1.6E-2	-	**2.8E-2**	1.4E-2	-	**1.2E-2**	1E-2	-	**1.9E-2**	1.1E-2	-	-	-

respectively; as such, DNN_{EF} would be a better solution for FPGAs with limited computing power.

Finally, table VIII compares our customized DNN_E with the conventional approach DNN_{base} in terms of throughput, memory footprint, and classification accuracy. The *Activity Recognition* dataset achieves 15× throughput improvement sacrificing only 0.1% classification accuracy. All other tasks are customized without loss of accuracy. On average, our customization results in 11.8× throughput improvement and reduces the memory footprint by a factor of 7.7×.

Table VIII: Outlined comparison between baseline (DNN_B) and customized (DNN_C) implementations.

Application	Accuracy (%)		Throughput (samples/s)		memory footprint (MB)	
	DNN_B	DNN_C	DNN_B	DNN_C	DNN_B	DNN_C
Speech Recognition	96	96	1057	11093	3.75	0.54
Indoor Localization	95.2	95.2	1168	12877	3.7	0.54
Activity Recognition	98.4	98.3	198	3204	20.23	1.8
MNIST	98.4	98.4	934	9137	5.44	0.71
Average	-0%	-0.025%	1×	11.8×	1×	0.13×

VI. Conclusion

This paper proposes a systematic design flow for platform-aware customization of deep neural networks executed on FPGA platforms. Our customization scheme employs a clustering-based algorithm to encode DNN parameters so that memory footprint and computational complexity are jointly reduced. Previous methods for customizing DNN models utilize fixed-point parameters. Our proposed encoding algorithm permits the use of floating-point parameters with low memory footprint, enabling automatic customization of DNNs for a broad variety of applications requiring different numerical precisions. Proof-of-concept evaluations on four different applications demonstrates up to 15-fold improvement in throughput and 9-fold reduction in memory footprint while the loss of accuracy remains below 0.1%.

Acknowledgment

This work was supported in parts by the Office of Naval Research grant (N00014-11-1-0885) and Trust-Hub grant (1649423).

References

[1] Krizhevsky *et al.*, "Imagenet classification with deep convolutional neural networks," in *NIPS*, pp. 1097–1105, 2012.

[2] Hinton *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *Signal Processing Magazine, IEEE*, vol. 29, no. 6, pp. 82–97, 2012.

[3] Deng *et al.*, "Recent advances in deep learning for speech research at microsoft," in *ICASSP, 2013*, pp. 8604–8608, IEEE.

[4] Mikolov *et al.*, "Recurrent neural network based language model.," in *Interspeech*, vol. 2, p. 3, 2010.

[5] Srinivas *et al.*, "Applications of data mining techniques in healthcare and prediction of heart attacks," *IJCSE*, 2010.

[6] B. Catanzaro, "Deep learning with cots hpc systems," 2013.

[7] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ACM Sigplan Notices*, ACM, 2014.

[8] C. Zhang, D. Wu, J. Sun, G. Sun, G. Luo, and J. Cong, "Energy-efficient cnn implementation on a deeply pipelined fpga cluster," in *ISLPED*, ACM, 2016.

[9] Lane *et al.*, "Can deep learning revolutionize mobile sensing?," in *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*, pp. 117–122, ACM, 2015.

[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 *FPGA*, ACM, 2015.

[11] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao, "Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks," in *FPGA*, ACM, 2016.

[12] 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*, pp. 32–37, IEEE, 2009.

[13] M. Denil, B. Shakibi, L. Dinh, N. de Freitas, *et al.*, "Predicting parameters in deep learning," in *NIPS*, 2013.

[14] W. Chen, J. T. Wilson, S. Tyree, K. Q. Weinberger, and Y. Chen, "Compressing neural networks with the hashing trick," *CoRR, abs/1504.04788*, 2015.

[15] Lin *et al.*, "Overcoming challenges in fixed point training of deep convolutional networks," *arXiv preprint arXiv:1607.02241*, 2016.

[16] Lin *et al.*, "Fixed point quantization of deep convolutional networks," *arXiv preprint arXiv:1511.06393*, 2015.

[17] Lin *et al.*, "Neural networks with few multiplications," *arXiv preprint arXiv:1510.03009*, 2015.

[18] M. Courbariaux, J.-P. David, and Y. Bengio, "Low precision storage for deep learning," *arXiv preprint arXiv:1412.7024*, 2014.

[19] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," *CoRR, abs/1502.02551*, 2015.

[20] Reagen *et al.*, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *Proceedings of ISCA*, 2016.

[21] M. Samragh, M. Imani, F. Koushanfar, and T. Rosing, "Looknn: Neural network with no multiplication," in *DATE*, IEEE, 2017.

[22] B. D. Rouhani, A. Mirhoseini, and F. Koushanfar, "Deep^3: Leveraging three levels of parallelism for efficient deep learning," in *DAC*, ACM, 2017.

[23] B. D. Rouhani, A. Mirhoseini, and F. Koushanfar, "Delight: Adding energy dimension to deep neural networks," in *ISLPED*.

[24] "Uci machine learning repository." http://archive.ics.uci.edu/ml/datasets/ISOLET.

[25] "Uci machine learning repository." https://archive.ics.uci.edu/ml/datasets/UJIIndoorLoc.

[26] "Uci machine learning repository." https://archive.ics.uci.edu/ml/datasets/Daily+and+Sports+Activities.

[27] Y. LeCun, C. Cortes, and C. J. Burges, "The mnist database of handwritten digits," 1998.

[28] F. Chollet, "keras." https://github.com/fchollet/keras, 2015.

[29] Pedregosa *et al.*, "Scikit-learn: Machine learning in python," *Journal of Machine Learning Research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[30] Sutskever *et al.*, "On the importance of initialization and momentum in deep learning.," *ICML (3)*, vol. 28, pp. 1139–1147, 2013.

[31] Srivastava *et al.*, "Dropout: a simple way to prevent neural networks from overfitting.," *Journal of Machine Learning Research*.