International Journal of Networking and Computing – www.ijnc.org, ISSN 2185-2847
Volume 11, Number 2, pages 172–197, July 2021

An implementation methodology for Neural Network on a Low-end FPGA Board[1]

Kaijie Wei, Koki Honda, Hideharu Amano
Dept. of Information and Computer Science, Keio University
3-14-1, Hiyoshi, Yokohama, Kanagawa, 223-8522, JAPAN

Received: February 13, 2021
Revised: May 5, 2021
Accepted: June 1, 2021
Communicated by Susumu Matsumae

Abstract

Artificial Intelligence(AI) has achieved unprecedented success in various fields that include image, speech, or even video recognition. Most systems are implemented on power-hungry devices like CPU, GPU, or even TPU to process data due to the models' high computation and storage complexity. CPU platforms do weak in computation capacity, while energy budgets and expense of GPU and TPU are often not affordable to edge computing in the industrial business. Recently, the FPGA-based Neural Network (NN) accelerator has been a trendy topic in the research field. It is regarded as a promising solution to suppress GPU in both speed and energy efficiency with its specifically designed architecture.

Our work performs on a low-end FPGA board, a more desirable platform in meeting the restrictions of energy efficiency and computational resource on an autonomous driving car. We propose a methodology that integrates a NN model into the board using HLS description in this paper. The whole design consists of algorithm-level downscaling and hardware optimization. The former emphasizes the model downscale through model pruning and binarization, which balance the model size and accuracy. The latter applies various HLS design techniques on each NN component, like loop unrolling, inter- /intra- level pipelining, and so on, to speed-up the application running on the target board. In the case study of tiny YOLO (You Only Look Once) v3, the model running on PYNQ-Z1 presents up to 22× acceleration comparing with the PYNQ's ARM CPU. Energy efficiency also achieves 3× better than Xeon E5-2667. To verify the flexibility of our methodology, we extend our work to the BinaryConnect and DoReFaNet. It is worth mentioning that the BinaryConnect even achieves around 100× acceleration comparing with it purely running on the PYNQ-Z1 ARM core.

Keywords: Neural Network, model compression, HLS streaming, FPGA, object detection

.

1 Introduction

Recent research on Neural Networks (NN) shows great potential over traditional algorithms in the field of Artificial Intelligence (AI) systems. Starting from Alexnet[18], which is considered as one of the most influential papers in computer vision, Convolutional Neural Network (CNN) broke into researchers' vision. Various CNN models like VGG[35], GooLeNet[38], ResNet[14] and so on continuously increase the top-5 image classification accuracy on the ImageNet dataset from

[1]The preliminary version of this paper was presented in the proceedings of the Eighth International Symposium on Computing and Network (CANDAR) 2020[19]

84.6% to 98.8%. Besides, these models are widely adopted in segmentation[24, 46, 47], object detection[33, 42, 55] and human pose estimation[5, 16, 36], which make CNN a promising candidate for many AI applications.

Table 1: Performance and architecture of state-of-art CNN models

Network model	AlexNet[18]	ResNet[14]	VGG-19[35]	DenseNet-264[15]	EfficientNet-L2[29]
Top-1 Accuracy	63.3%	75.9%	72.7%	77.9%	90.2%
Top-5 Accuracy	84.6%	92.9%	91.0%	93.9%	98.8%
Num of params	60M	25M	144M	8M	480M
Depth	8	50	19	264	stochastic
Year	2012	2016	2014	2016	2021

However, the computational resources and the storage complexity come to be burdens on platforms, as you can notice from Table 1. We present the representative model performance of top-1 and top-5 accuracy evaluated on ImageNet dataset [10], the number of parameters, model depth, and the year that paper delivered. Take the latest model, EfficientNet-L2 [29], as an example, for a 475×475 image classification, it requires up to 37 billion floating-point operations (FLOP) and more than 480MB parameters.

A general CPU can perform 10 to 100 GFLOP/s at the power efficiency below 1GOP/J, so CPUs can neither meet the data-processing requirements in high-performance computing (HPC) nor the low-power requirements in a mobile application. Hence, the CNN applications usually run on devices like the cloud, high-end FPGAs, or GPUs. Among them, the most popular one is GPU. It offers up to 10TOP/s peak performance and is the first choice of NN application. Nonetheless, none of these platforms are approachable to a class of edge devices due to their high cost and energy consumption. On the other hand, although ASICs of domain-specific architectures can achieve high performance with low power consumption [45], its flexibility still tends to be limited. Considering flexibility and power efficiency, FPGA can be a better choice to implement a model in high parallelism without hurting the model accuracy. To confront the requirements of edge devices like autonomous cars directly, we choose a low-end FPGA board as our platform, considering its limited resources in addition to the cost of manufacture. Though low-end FPGAs have been a hopeful platform, there still exist several challenges.

1. Challenges in general FPGA-based NN accelerator
    - There are no supported NN frameworks like Tensorflow or Pytorch for GPU or CPU, so it is much harder to implement NN on FPGAs than that on CPUs or GPUs.
    - Current FPGAs work at the frequency of 100-300MHz, which is much less than CPU or GPU. Besides, the logic overhead for reconfigurability also reduces the overall performance. Thus, a straightforward design on FPGA is hard to achieve high performance
2. Challenges of NN implementation on a low-end FPGA board
    - It is quite difficult to embed a resource-consuming NN model into a resource-limited hardware device with considering the memory capacity.
    - To use resources efficiently, a highly optimized design should be implemented.

To use resources on-board efficiently, a lot of researches using Hardware Descriptive Language (HDL) has been proposed and reached state-of-art performance[30, 1]. However, in our design, we adopted the High-Level Synthesis (HLS) approach instead. There are mainly 2 reasons for this choice. First, from the viewpoint of the recent speedy advancement of CNN algorithms, the HDL design is not advantageous for introducing newly developed skills. Second, considering the development of the target devices, a quick exploration of the performance and cost is required. From these 2

aspects, the HLS design is much more advantageous. Although several implementations using HLS have been proposed[34, 44, 25, 12, 48], most of them adopt high-end FPGA boards, which are hard to be approached in edge computing. FINN[3, 39], an HLS implementation, which is proposed by Xilinx Research Labs, has been extensively applied by FPGA developers who have interests in NN development. Nevertheless, their mechanism pays more attention to the hardware design and works on the premise that the input data has been fully quantized.

Our methodology organized in the following steps focus on both model compression and hardware optimization,

1. Revise the network architecture through network model pruning to downscale some high-end NN models.

2. Binarize part of the NN model to further reduce the memory capacity and make the model much more suitable for FPGA[7].

3. Design the modules of NN components through Vivado HLS Design suite and arrange Intellectual Properties (IPs) of the whole system upon Zynq architecture.

During the experiment, we choose one of the low-end FPGA boards called PYNQ-Z1, a Python-based Zynq board that provides a user-friendly programming environment. It uses Jupyter notebook as the interface to use the hardware libraries and overlays on the programmable logic.

As for the test case, we choose You Only Look Once (YOLO) v3[32], an end-to-end fully connected CNN architecture. Its characters, like high accuracy performance, unified kernel size, and few floating-point operations with high computation speed, benefit later FPGA implementation. YOLO v3 is also named Darknet-53, which contains 53 convolutional layers, each followed by batch normalization and Leaky ReLU activation. Considering the limitation of our platform, a variant of YOLO v3, YOLOv3-tiny, comes to be our choice.

This model is approximately 442% faster than its larger big brothers with its compact size($<$ 50MB) and faster inference speed with the sacrifice of accuracy, which makes it much more suitable for embedded systems[28]. Max pooling layers included in the model can efficiently downsample FMs.

The rest of the paper is organized as follows, Section 2 gives background and terminology that relates to this work. Section 3 proposes our methodology and implementation of our design, including algorithm-level downscaling and hardware optimization. We evaluate our design in Section 4. Finally, we review our paper and present future work in Section 5.

2 Background and Terminology

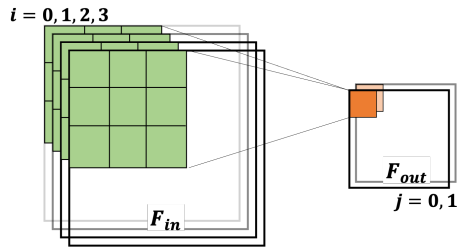
Before discussing the methodology, we first introduce some basic concepts and terminologies used in CNN. Besides, model compression techniques adopted in this paper, which is applied to reduce the computation and memory complexity, will also be demonstrated. Finally, the last subsection will highlight the challenges of implementing an NN model on an FPGA board with visiting some related work.

2.1 Neural Network Components

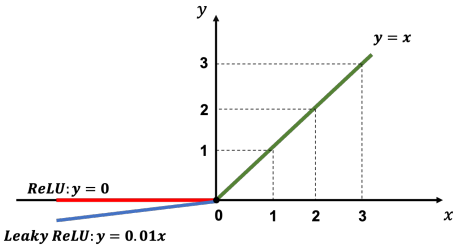
A common form of CNN architecture involves sets of convolutional layers, the ReLU layer, followed by pooling layers. This pattern repeats until all features are extracted, and the image has merged spatially to a small size. Finally, the fully connected layers are adopted to detect objects in the image.

2.1.1 Convolution(CNV) Layer

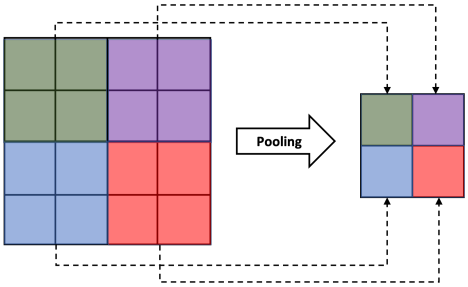
The convolution layer, adopted in all CNN models, is the kernel part of the NN modeling. It is mainly used to detect the features in a feature map(FM). The output volume of the convolutional



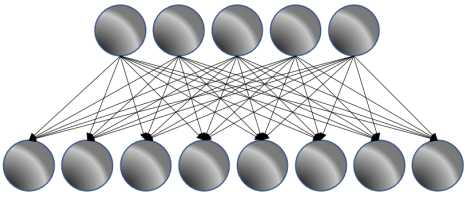
(a) Convolutional operation



(b) Activation functions



(c) Pooling functions



(d) Fully connected layer

Figure 1: Neural Network components

layer is obtained by stacking the activation maps of all filters along the depth dimension M. The convolutional operation, which denotes as the function $conv2d$ in Eqn. 1.

$$F_{out}(j) = \sum_{i=1}^{M-1} conv2d(F_{in}(i), K_{ij}) + b_j \tag{1}$$

As is shown in Figure 1a, the CNV layer performs 2D convolution on a set of input FMs F_{in} and sums up the results to get output FM F_{out}. i in the formula represents the channel of the input FMs. j stands for a channel of output FMs. $k_{i,j}$ is a filter kernel connecting the i^{th} channel of the input volume and the j^{th} channel of the output volume. And b_j donates the bias term of the j^{th} output channel.

2.1.2 Activation Functions(AFs)

AFs are one of the crucial components of NN models. They are mathematical equations that determine the output of the NN. The non-linearity of functions is responsible for an increased degree of freedom of the learner, enabling them to generalize problems from high dimension to lower ones, with data to differentiate between outputs. Two popularly used AFs present in Figure.1b. One is the ReLU function represented as

$$ReLU(x) = max(x,\ \ 0) \tag{2}$$

where x represents the weighted sum of the neuron inputs. The function allows the network to converge very quickly. However, the issue in the ReLU function is called "Dying ReLU" that all the negative values become zero immediately. The problem decreases the ability of the model to fit or train from the data properly. To settle such a problem, the Leaky ReLU AF, which is defined as

$$LReLU(x) = max(x,\ \ 0.01x) \tag{3}$$

was proposed. As can be noticed from Figure 1b, this function will not provide consistent predictions for the negative input value.

2.1.3 Normalization

Normalization, which changes the value of numeric columns into a data-set using a common scale, is an approach to apply during the propagation of an NN, while features in the model are in different ranges.

The most commonly adopted method is Batch Normalizatio(BN), which stabilizing the distribution of layer inputs during the training phase. It enables a faster and far more stable training of a CNN model. On the contrary, the Shift-based BN(SBN) is more preferred in the binarized models due to its significant reduction in computing resource cost at a loss of precision, which is more friendly to the hardware implementation than the conventional one. The forward propagation of BN and SBN are explained in Algorithm 1. The BN trains neurons over a mini-batch through 4 steps, including mean, variance, normalization, and scale-and-shift. Instead, the variant version of BN, the SBN, is composed of one more step to get the centered input, and the following operations are all based on this value. To deal with a large number of multiplications in BN, shift and $AP2$ are adopted instead. The hardware implementation of $AP2$ is as simple as extracting the index of the most significant bit from the number's binary representation, which saves large amounts of computational resources in the hardware design comparing with the conventional one.

Algorithm 1 Algorithm for BN and SBN[17]

Function: $AP2$ stands for the approximate power-of-2
$\ll\gg$ indicates both left and right binary shift
Input: Value of X over a mini-batch: $\mathcal{B} = x_{1...m}$
Parameters to be learned: γ, β
Output: $\{y_i = BN_{\gamma,\beta(x_i)}\}$

$\mu_{\mathcal{B}} \longleftarrow \frac{1}{m}\sum_{i}^{m} x_i$ // mini-batch mean

$C(x_i) \longleftarrow (x_i - \mu_{\mathcal{B}})$ // centered input(Only in SBN)

$\sigma^2_{\mathcal{B}} \longleftarrow \frac{1}{m}\sum_{i+1}^{m}(x_i - \mu_{\mathcal{B}})^2 \Longrightarrow \sigma^2_{\mathcal{B}} \longleftarrow \frac{1}{m}\sum_{i}^{m}(C(x_i) \ll\gg AP2(C(x_i)))$ // mini-batch variance

$\hat{x}_i \longleftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \Longrightarrow \hat{x}_i \longleftarrow C(x_i) \ll\gg AP2((\sqrt{\sigma^2_{\mathcal{B}} + \epsilon})^{-1})$ // normalize

$y_i \longleftarrow \gamma\hat{x}_i + \beta \equiv BN_{\gamma,\beta}(x_i) \Longrightarrow y_i \longleftarrow AP2(\gamma) \ll\gg \hat{x}_i$ // scale and shift

2.1.4 Pooling Layer

Pooling, also known as subsampling, is manipulated to reduce the spatial size of the representation. Furthermore, it eliminates the number of parameters and computation in the model. There are mainly two types of pooling, **Max pooling**, and **Average pooling**. As shown in Figure.1c, in the portion of the image covered by the kernel, the former returns the maximum value of a covered FM, while the latter returns the average of that[20]. For instance, the most common form is a pooling layer with filters of size 2×2 applied where the MAX operation will be taken over 4 samples, thereby discarding 75% of the activations.

2.1.5 Fully Connected(FC) Layer

The fully connected layer is an essential component of CNN, which has been widely used in recognizing and classifying an image. Generally, it locates in the last part of an NN model. The input layer, the upper part of Figure 1d, takes the output of previous layers, flattens them, and turns them into a single vector that can be an input for the next stage to apply weights and predict the correct label. Finally, the output layer gives the final probabilities for each label.

2.2 CNN Compression

In this subsection, we will explain the techniques to compress a CNN model. They reduce the computation and memory footprint of a large NN model at minimal sacrifice on the accuracy. In this paper, two adopted techniques to compress the NN model, layer-based pruning and model binarization, will be explained later in detail.

2.2.1 Model Pruning

As discussed in [23], pruning works at different levels including weight level, channel level, layer level, and so on. The weight level pruning, which is regarded as fine-grained level sparsity, provides the highest flexibility and leads to a higher compression rate. However, since the algorithms prune the weights individually and the pruned structures are unstructured, they usually require specific algorithms to conduct fast inference on the pruned models. On the contrary, coarse-level pruning, which prunes at the level of filter, channel, or layer, can efficiently avoid such a problem at the cost of flexibility, since the original convolutional structure is still preserved. Considering the upcoming hardware optimization, in our methodology, we adopt channel pruning in our methodology.

We follow Algorithm 2, which is introduced in subsection 2.2.1 to prune a model. First, an NN architecture is defined as a function-family $f(x;\cdot)$ to produce outputs from inputs, which includes the NN components explained in Subsection 2.1. Then, W represents some specific parameters in the model. NN pruning takes a model $f(X;W)$ as input and produces a new model $f(x;W')$. Here, W' is a set of parameters that may be different from W. λ is to balance empirical loss and sparsity. Finally, the γ is a scaling factor for each channel.

Algorithm 2 Algorithm for Pruning and Fine-tuning

Parameter: scaling factor for each channel γ
λ balances the empirical loss and sparsity-induced penalty on γ
Input: N, the number of iterations of pruning
X, the dataset on which to train and fine-tune

1: $W \Leftarrow initialize()$
2: **for** $i = 1$ `to` N **do**
3: $W \Leftarrow SparseRGL(f(X;W),\lambda)$ // Train with channel sparsity regularization
4: $f(X;W') \Leftarrow prune(f(X;W),\gamma)$ // Prune channels with small scaling factors
5: $W \Leftarrow fineTune(f(X;W'))$ // Fine-tune the pruned network
6: **end for**
7: **return** W

2.2.2 Model Binarization

Considering the limited resources on the platform and the complexity of an NN model. The most extreme form of network quantization, binarization, is adopted. It is a 1-bit quantization, which can only have 2 possible values, -1 and +1[8]. We follow the Algorithm 3 to train the model in binary. The straight-through estimator (STE)[8] is accepted with considering the saturation effect. Besides, the deterministic is preferred rather than the stochastic one.

2.3 Challenges of FPGA-based NN Implementation and Related Work

Implementing an NN on an FPGA includes several challenges. First of all, the requirement of significant storage and computational resources. For example, Guo et al.[13] proposed an FPGA implementation on Zynq XC7Z045, which quantizes the VGG-16 and parallel PEs on their design. As a result, it consumes 89% BRAMs, 84% LUTs, and 87% DSPs on the platform. [26] is also a methodology working on Altera Arria 10. It pays more attention to the CNN loop optimization by exploring the trade-offs of hardware and proposing a specific data flow to minimize the data access

Algorithm 3 Algorithm for Model Binarization

Parameter: the number of layers L, the learning rate decay factor λ

Function: the cost function for minbatch C, element-wise multiplication $\circ$, $Binarize()$ binarizes the activation and weights, $Clip()$ clips the weights, $BatchNorm()$ batch-normalizes the activations, $BackBatchNorm()$ backpropagates through the normalization, $Update()$ updates the parameters when the gradients are known

Input: a minibatch of inputs and targets (a_0, a^*), previous weights W, previous BatchNorm parameter θ, weight initialization coefficients γ, previous learning rate η

Ensure: updated weights W^{t+1}, updated BatchNorm parameters θ^{t+1}, and updated learning rate η^{t+1}

// Forward propagation:
for $k = 1$ `to` L **do**
  $W_k^b \Leftarrow Binarize(W_k)$ // Deterministic Binarization
  $s_k \Leftarrow a_{k-1}^b W_k^b$
  $a_k \Leftarrow BatchNorm(s_k, \theta_k)$ // SBN[8] (which simply replaces the costly multiplication with a shift at the cost of accuracy)
  if $k < L$ **then**
    $a_k^b \Leftarrow Binarize(a_k)$
  end if
end for
// Backward propagation:
for $k = L$ `to` 1 **do**
  if $k < L$ **then**
    $g_{a_k} \leftarrow g_{a_k^b} \circ 1_{|a_k| \leq 1}$ // $g_{a_L} = \frac{\partial C}{\partial a_L}$ knowing a_L and a^* are not binary
  end if
  $(g_{s_k}, g_{\theta_k}) \Leftarrow BackBatchNorm(g_{a_k}, s_k, \theta_k)$
  $g_{a_{k-1}^b} \Leftarrow g_{s_k} W_k^b$
  $g_{W_k^b} \Leftarrow g_{s_k}^\top a_{k-1}^b$
end for
// Accumulating parameters gradients:
for $k = 1$ `to` L **do**
  $\theta_k^{t+1} \Leftarrow Update(\theta_k, \eta, g_{\theta_k})$ // Shift-based AdaMax[8]
  $W_k^{t+1} \Leftarrow Clip(Update(W_k, \gamma_k \eta, g_{w_k^b}), -1, 1)$ // $Clip(x, -1, 1) = max(-1, min(1, x))$
  $\eta^{t+1} \Leftarrow \lambda \eta$
end for

and movement. [43] proposed an implementation flow that provides us with a deeper insight into XNOR gate implementation. Instead of conventional BNN implementation using XNOR gate like our design, they modify the architecture of an NN model through logic expansion and generate LUTNet to realize a higher logic density in the upcoming hardware-level implementation. Although all of these works achieved state-of-art FPGA implement of the CNN model, there are far fewer resources available on the target board than those used in these researches. Therefore, it is incomparable to their work. Instead, we compare our work to [41, 11, 40], which perform their work on middle/low edged devices.

Second, different layers in a CNN model have their distinctive characteristics, which result in various forms of parallelism and memory access. Regarding the varying requirements of different layers, we should design an accelerator over the whole CNN model. Researches like [50] and [49], only emphasize the convolutional layer design without considering other components in the CNN model.

Finally, a mechanism to reuse computational resources and store the partial result in on-chip memory is also crucial to the implementation, which will result in a better performance on energy

efficiency and acceleration. Here, we will also compare our work with [37, 53, 51, 6], which take great advantage of this point in their design.

3 Proposed Methodology

As is shown in Figure 2, there are 2 different phases proposed in our methodology. The algorithm-level downscaling aims at the model compression and the hardware optimization corresponds to HLS description/directives to find the best execution speed with limited resources.

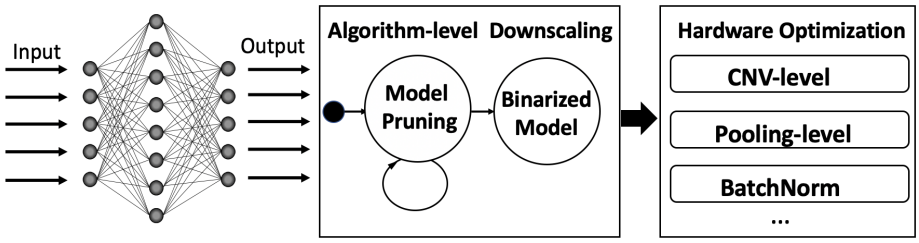


Figure 2: Proposed Methodology

3.1 Algorithm-level Downscaling

Regarding the algorithm-level downscaling, mainly two stages are performed: model pruning and model binarization, which is introduced respectively in section 2.2.1 and section 2.2.2. For the hardware optimization, instead of HDL implementation to tailor the given algorithm, we choose HLS based design in our methodology to maintain and update the model in a more productive way.

There is a trade-off between the means Average Precision (mAP) performance[2] and execution speed. The mAP is an accuracy metric of object detection and execution speed corresponds to model compression and hardware optimization. There will be more space for enhancing the execution speed on the resource-limited board with the decreased resources. The key idea is to balance the obtained performance and resource utilization by trying various designs with HLS iteratively. Our final goal is to establish a design flow to find the balance between mAP performance and execution speed on the target FPGA board by applying various techniques on algorithm-level downscaling and hardware optimization iteratively. To demonstrate our design, we first choose YOLO v3[32] as our target NN model, and PYNQ-Z1[4] is our low-end FPGA platform. We will illustrate these 2 stages and implementation details carefully in the following subsections.

3.1.1 CNV Pruning

Before pruning the model, we attempt to substitute the Leaky ReLU with the ReLU function for mainly 2 reasons. First of all, considering the computational complexity, ReLU is less complex than any other AF[27], which can save some computational resources on the device, as well, speed up the computation. Then, from the perspective of performance, although we may encounter the problem of “dying ReLU", in the following process of binarization, the model will go through the STE to activate the neuron. The functions will migrate the seriousness of the problem.

Based on [23], which is generally explained in subsection 2.2.1, the pruning function is represented by

$$L = \sum_{(x,y)} l(f(x, W), y) + \lambda \sum_{\gamma \in \tau} g(\gamma) \tag{4}$$

where (x, y) is the training input and target, W denotes the trainable weights, $g(\cdot)$ is a sparsity-induced penalty on scaling factors, and λ balances the two terms. Finally, the γ is a scaling factor for each channel.

We present the flow-chart of the NN pruning procedure in Figure 3. The dotted line stands for the multi-pass/ iterative scheme. The following will detail our processes for pruning the NN models.

First, in the initial step of the pruning, we performed **original model training** from scratch as the baseline. Regarding the YOLO on COCO dataset, we train the model on the mini-batch size of 64 for 231 epochs on SGD. We initialize the learning rate as 0.01 and divide it by 10 at 50% of the total number of training epochs. The weight decays at a rate of $1e^{-6}$ to avoid the problem of overfitting. The momentum of 0.9 is to accelerate the training. Besides, according to [23], the scaling factor is first initialized as 0.5, since it gives a higher accuracy for the baseline models without any pruning process.

Following the initial training, the trained model works with **channel sparse regularization**. The key to this process is to find the hyperparameter λ, which controls the trade-off between empirical loss and sparsity. For the YOLO on the revised COCO dataset, we set $\lambda = 1e^{-4}$. All other settings are the same as in previous training. After training under sparse regularization, we obtain a model in which many scaling factors are near zero.

After the training with channel sparsity regularization, we eventually obtain a model, which many scaling factors are near zero. To prune channels with near-zero scaling factors, we remove their related connections and weights. Instead of pruning different layers with different ratios, we use a global pruning threshold to simplify our implementation. The pruning ratio is determined among all the scaling factors, which are initialized as 0.5 and will be leveraged in the BN layers. In the case of YOLO, the pruning ratio of 86%, which means the fine-tuned model with 86% channels are pruned from the model trained with sparsity.

In the last step, on the compact model, **fine-tuning** trained with the learning rate of 10^{-3} for 7 epochs. Other optimization settings are kept unchanged.

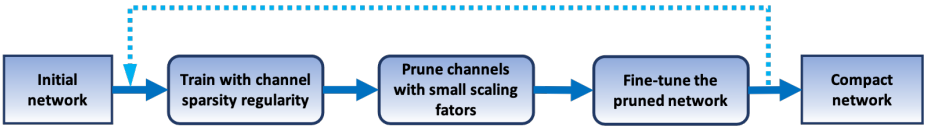


Figure 3: Flow-chart of NN pruning procedure

3.1.2 Model Binarization

In this stage, we binarize both weights and activations in the hidden layer. The reason for the input/output layer is not binarized is as follows: According to [31, 54, 22], if layers, which do not have a shortcut connection, are binarized, the information is lost due to the binarization. The lost information can never be recovered from the subsequent layers of the network. This affects the first input layer and the last output layer. In other words, both of these two layers take the most significant status in the whole model, and it is not rewarding to binarize the input layer considering the mAP degradation. This process reduces the resource utilization and computation complexity through the use of bitwise operations. We present the procedures of model binarization in Figure 4. The convolutional function in BNN is calculated as XNOR and bit-count operations followed by max pooling and BN. We performed neuron binarization through an AF called “deterministic function". It will be introduced in subsection 3.2.2. These 4 steps iterate until they go through the whole pruned model.

Furthermore, we deduce that a high learning rate will lead to frequent weight sign change. During the training, we keep the learning rate below 0.001. Finally, according to [8], applying max pooling on the binary input returns a tensor (matrix) whose most of its elements are equal to +1, resulting in a noticeable drop in mAP performance. Thus, in our design, we put the max pooling layer before the BN and activation.

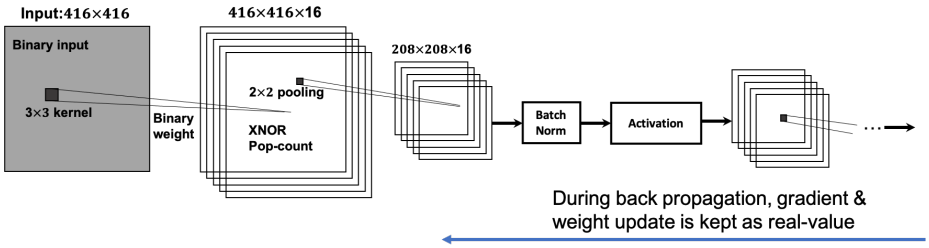


Figure 4: The training of a BNN

3.2 Hardware Optimization

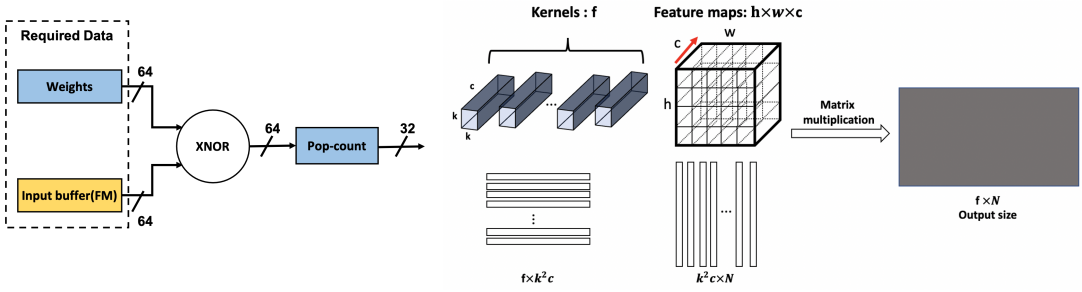
The hardware optimization works on the PYNQ-Z1 board, a Python-based ZYNQ system. The application on PYNQ-Z1 runs on the dual-core cortex-A9 and uses Jupyter Notebook as a user-interface. The key specifications of this board are listed in Table 2. As can be noticed from the table, the implementation of YOLO v3 is a heavy burden on the resource of LUTs, Block RAM, and DSP Slices.

Table 2: Key PYNQ-Z1 Specifications

Components	Size
Logic slices	13,300
6-input LUTs	53,200
Flip-Flops	106,400
Block RAM	630 KB
DSP Slices	220

In the following subsections, we will discuss the designed module of each component in CNN, which we introduced in subsection 2.1. Subsection 3.2.1 corresponds to the CNV design in hardware, and subsection 3.2.3 refers to the max pooling layer. Besides, we also introduce the activation and BN-level optimization in subsection 3.2.2 as well as the FC implementation in subsection 3.2.4.

3.2.1 CNV-level Optimization



(a) CNV computation in PE

(b) Convolutional operation column expansion

Figure 5: CNV-level Optimization in module design

Since the model is fully binarized, the binary value can either be -1 or +1. We refer to these signed values as binary values 0 and 1. As presented in Figure 5a, it describes the design of CNV-operation in a Processing Element(PE). Taking advantage of BNN, we can use the XNOR gate to substitute

the multiplication of weights and FM in the input buffer. Such substitution can remarkably simplify the computation. Regarding the accumulation of binary dot-product, we use pop-count design, the implementation of counting the number of set bits (1), to accumulate the set bits from the XNOR instead. These substitutions can efficiently save the resource of DSPs and FFs, comparing with the computation of signed Multiplication-and-Accumulation (MAC).

When theFMs are transferred to buffer, they are all converted to column vectors. Each column vector represents a block of the FM, which will perform the XNOR gate operation with a kernel as shown in Figure 5b. Besides, to decrease the latency, the "interleave" operation is also performed in our design. Conventionally, the dimension of a FM is represented as (batch size, input channel, height, and weight), while that of a kernel is represented as (Output channel, input channel, height, and weight). Considering such dimension order, if we transform these matrices into datastream (AXI4-Stream), we need to save almost the whole FM. With considering the pop-count operation, as long as the vectors correspond to the unrolling order, the order of a column will not affect the output result. As a result, these dimensions are transposed, and the input channel becomes the last dimension, and data is streamed in the direction of channels with interleaving. This implementation significantly lessens the burden of on-board memory.

After expanding the shape of the input data, we parallelize the convolutional operation. FMs from the input buffer passes to the Processing elements (PEs), and the result transfers to an output buffer. We applied both inter- and intra- level pipelining to the basic computation. As shown in Figure 6, in the intra-level pipeline, there are three stages: load, compute, and store. They enable higher-level parallelism for enhancing the throughput. To begin with, in the "load" stage, a part of FMs are loaded to buffer. In the "compute" stage, the operations are further parallelized using the technique called inter-level pipelining. We set the maximum stages for each kernel data as 32 which means that most 32 pairs of operands can be calculated simultaneously. Since it is working on a binarized model, it assumes that there will be a high utilization in LUT. Finally, the results pass to the final stage of the intra-layer pipeline, and then they are stored in the output buffer.

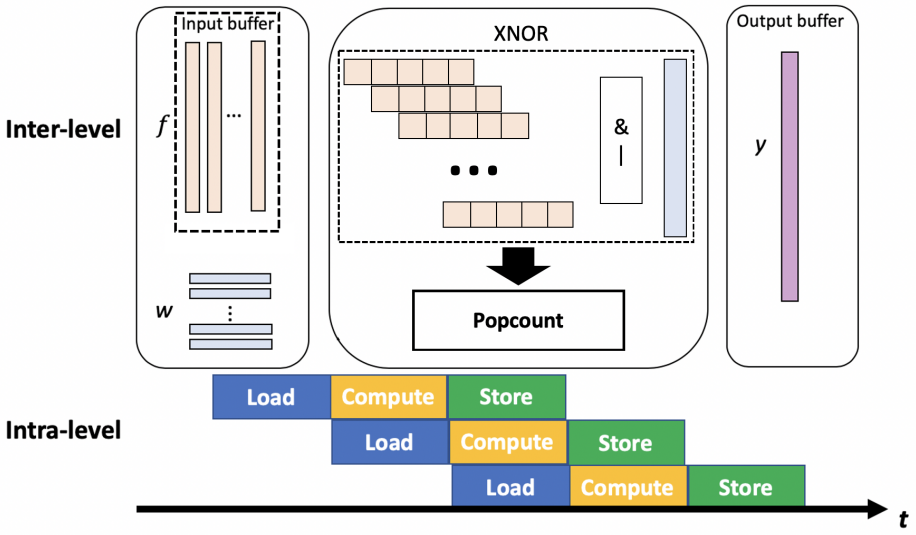


Figure 6: Pipeline in CNV-level

3.2.2 Activation and BN-level Optimization

According to Algorithm 1, BN can be presented as Eqn.5, where μ stands for the mean value, σ denotes the standard deviation, ϵ is added for stability, γ, and β are the learned values to implement

scale and shift for an identity transformation.

$$BN(x) = \frac{x-\mu}{\sqrt{\sigma^2+\epsilon}} * \gamma + \beta \tag{5}$$

Since the conventional BN seems to be a costly computation, as can be noticed from Eqn.5, floating-point multiplications are required for every normalization step, which will lead to a heavy burden on the DSP and other memory resources. Thus, we introduce the SBN, as is introduced in subsection 2.1.3. Correspondingly, it can be calculated as Eqn.6, where $\phi = round(\log_2 \left|\frac{\gamma}{\sigma}\right|)$ is the left-shift value for both σ and γ. $sal(x, y)$ is an function to left shift to x by y bits.

$$SBN(x) = sal[(x-\mu), \phi] \cdot sign \left|\frac{\gamma}{\sigma}\right| + \beta \tag{6}$$

In this way, comparing with the conventional BN, all the multiplication are replaced with shift operations. We first calculate the required parameters for BN like $\frac{\gamma}{\sigma}$ and store them into the corresponding cache.

Following SBN, an activation layer is applied to activate the neurons in the model. Implementing these functions in the forward pass can be equivalent to an integer threshold in a binarized model. The activation layer is simply calculated as $Sign(SBN(x))$ [8].

$$Sign(y) = \begin{cases} +1, & \text{if } y \geq \tau \\ -1 & \text{otherwise} \end{cases} \tag{7}$$

where

$$\tau = \frac{-\beta\sqrt{\sigma^2}}{\gamma} + \mu \tag{8}$$

Besides, since the activation layer works on the Eqn. 7, the result is determined by comparing the input with a threshold τ[8]. In our BNN model, we use the dot product to produce integer output, which means the τ can be rounded approximately to simplify the computation. This method is quite suitable for the FPGA implementation because the thresholding is much simpler than arithmetic computation on the logic design.

3.2.3 Pooling-level Optimization

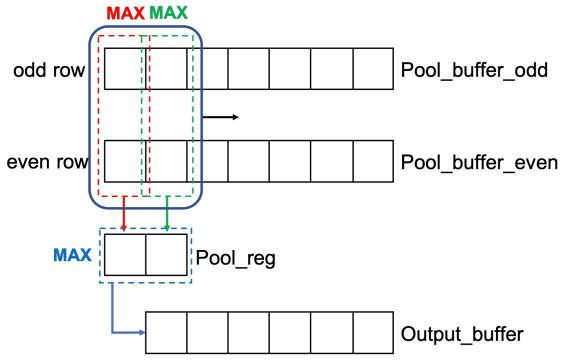


Figure 7: Hardware Optimization of Max Pooling

Since we place the pooling layer directly after the pop-count, a pooling layer gets a signed integer instead of a binary value. In our design, we focus on the implementation of max pooling, which applies to YOLO v3. We prepare two line-buffers of the same size as presented in Figure 7. The upper one loads the odd-numbered rows of data, and the other one saves the data from even-numbered rows of data.

The pooling is divided into two different stages. In the first stage, we compare the corresponding elements in buffers at the same time. The larger one is retained for the next step and save in the register. In the last stage, when 2 intermediate results are both obtained, the larger one is selected as the output of the max pooling size of 4 and written to the output buffer. Such a design is also suitable for the average pooling. As the design in CNV-level optimization, pooling windows are also processed in parallel to match the throughput of CNV-level optimization.

3.2.4 FC-level Optimization

FC layers is represented as $a_N = A \cdot a^b_{N-1}$ where A is the weight matrix and a^b_{N-1} are the activated output from the previous layer. Since the output of FC also needs to be activated, we can compute the activated output through $a^b_N = a^N > \tau^+_N$[39]. τ^+_N itself is fixed for a trained NN and can derive from BN parameters at a compile-time. Therefore, the FC-level optimization is regarded as the matrix-Vector-Threshold (MVTU) problem proposed in [39]. We imported the library of their MVTU design during the implementation of our FC-level optimization. The design consists of an input and output buffer and an array of PEs, each with several SIMD lanes.

3.3 Overall Hardware Architecture

As is introduced previously, we embed our system on the PYNQ-Z1 board, the overall hardware bases on this ZYNQ board architecture. The board is a platform mainly for PYNQ open-source framework. The Programmable Logic (PL) takes the most significant place during the acceleration of the whole system. It mainly consists of elements that we include in Table 2. The software runs on 650MHz dual-core Cortex-A9 CPUs, which is called the Processing System (PS). It includes a web server hosting the Jupyter Notebook design environment, the Ipython kernel and its packages, Linux, and the base hardware library API for FPGA.

The left part of Figure 8 represents the ARM core, which runs on Linux Ubuntu. It realizes data transmission of instructions and convolutional data between software PE and PL part. Since the input layer and output layer are not binarized, both of them work on this part. Besides, there are some exceptions. For example, YOLO accepts bounding boxes to detect objects instead of a cascade of FC layers. The bounding boxes are all implemented on this part.

The right one shows the PL design on the board. It consists of direct memory access (DMA), controller, input/output buffer, and PEs. It takes charge of all the operations introduced in subsection 3.2. Since we binarized hidden layer of the model, unfortunately, we encounter the LUT shortage. As a strategy, we move part of the logic calculation from LUT to DSP48 by returning the binarized data to 16-bit floating-point instead of the original 32-bit ones so that the model can be embedded into PYNQ-Z1 successfully. Finally, the convolutional result of the hidden layer returns to the PS part through DMA. The PS part begins its work of output layer.

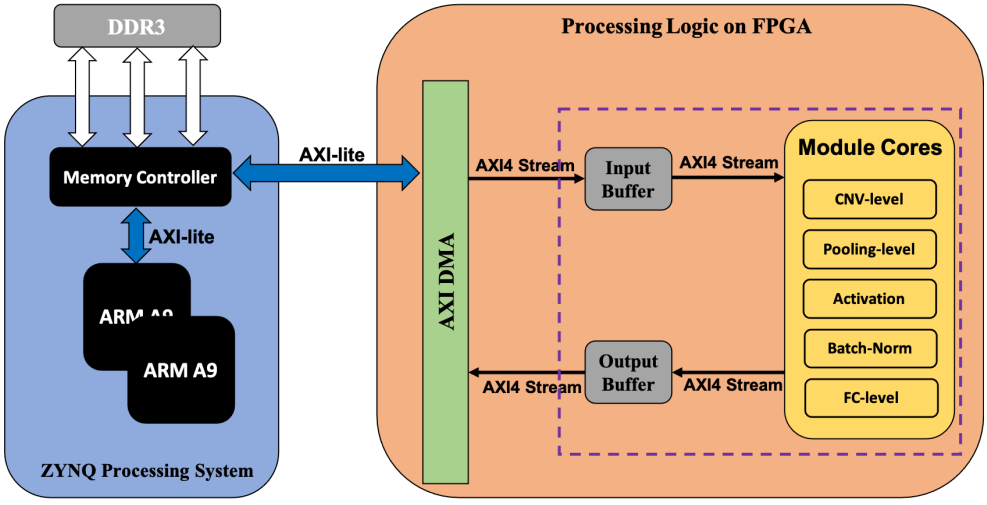


Figure 8: Overall hardware architecture

4 Evaluation

4.1 Algorithm-level Downscaling

We use Tesla P100 to perform the training. Since the model mainly works on some low-end edge devices, it is not necessary to use the complete version of the COCO dataset[21]. We constrain the COCO dataset to 8 classes including people, bicycles, dogs, and so on.

4.1.1 CNV Pruning Evaluation

Before pruning, as introduced in subsection 3.1.1, we substitute the AF in the model with ReLU. The revised model's mAP is about 94.91%. There is only a 0.41% decrease in precision comparing with the original model. Although this technique shows little effect on the model size, the degradation is quite rewarding, considering the reduction of computation complexity.

We followed the steps represented in subsection 3.1.1. We trained the network model through channel-sparse regularization, pruning, as well as fine-tuning. The pruned result lists in Table 3. To be more specific, the pruned model and its pruning details present in Table 4, which includes the pruning rate of each layer. From the table, we can also notice that although we have set 86% channels to be pruned, it is not exactly 86% channels of each layer pruned.

Table 3: Pruning result

Steps	Parameter	Model size	Inference	mAP
Input	31.5M	171.4M	8.0ms	94.91%
Pruning	5.5M	28.2M	4.2ms	94.93%
Finetune	5.5M	28.2M	4.2ms	94.94%

Table 4: Pruned model and Pruning details

	Original model (layers)	Pruned model (layers)	Pruning rate
CNV1	16	16	0.000%
CNV2	32	19	40.625%
CNV3	64	25	60.938%
CNV4	128	51	60.156%
CNV5	256	47	81.641%
CNV6	512	31	93.945%
CNV7	1024	122	88.086%
CNV8	256	64	75.000%
CNV9	512	50	90.234%
CNV10	255	96	62.353%
CNV11	128	16	87.500%
CNV12	265	41	83.984%
CNV13	255	59	76.863%

Furthermore, regarding the YOLO model, to investigate the effectiveness of the pruning algorithm, which the pruning ratio is 86%, we further retrain the model with the pruning ratio from 70% to 90% to explore the mAP precision variation. The evaluation includes the last 2 steps of the pruning procedures, pruning as well as fine-tuning. As can be noticed from Figure 9, fine-tuning is quite effective on the pruning ratio around 70% and 80%. When it comes to 85%, the pruning reaches a status of saturation, and the fine-tuning does not work very well. As we can notice from Table 5, when the pruning ratio is larger than 87%, the network model is seriously damaged, and the mAP performance deteriorates dramatically. Thus, we can summarize that the 86% of pruning ratio is the limit to this YOLO model.

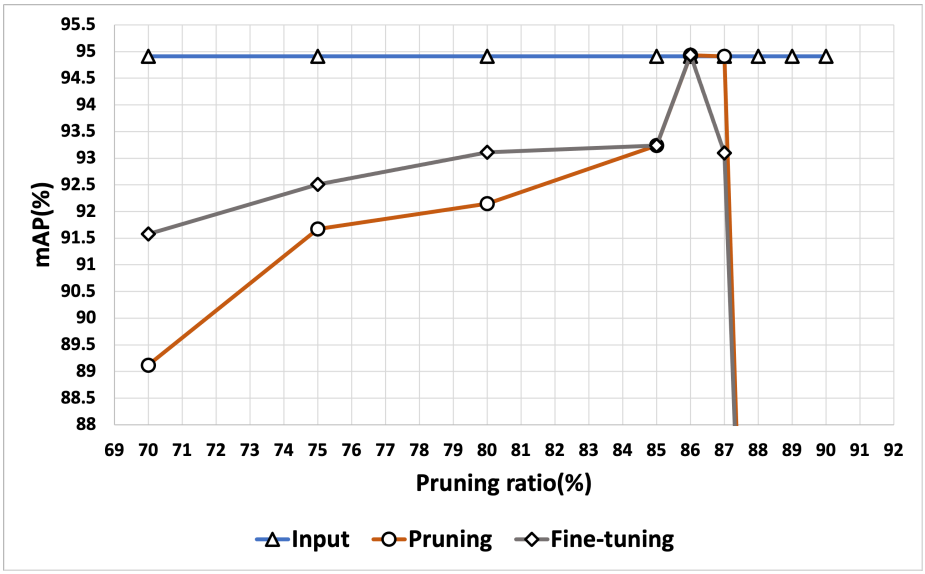


Figure 9: Pruing evaluation under different pruning ratios

Table 5: Pruning result

Pruning ratio / mAP(%)	70	75	80	85	86	87	88	89	90
Input	94.91	94.91	94.91	94.91	94.91	94.91	94.91	94.91	94.91
Pruning	89.12	91.67	92.15	93.24	94.93	94.91	75.22	20.45	12.1
Fine-tuning	91.58	92.51	93.11	93.24	94.94	93.1	76.31	20.21	10.15

To conclude, through the adopted pruning algorithm, the mAP precision increased by 0.02%. There are 82.5% parameters pruned, which result in the model size decreased by 83.55%. Because of the compact model, the inference time reaches 2 times the input model. However, we should also point out that the pruning algorithm copes with some CNNs with multiple layers in a high efficiency while not with some thin network models.[23]

4.1.2 Model Binarization Evaluation

As is introduced previously, only the hidden layer are binarized. Since this stage is the last step of the model revision, we present the IoU(Intersection of Union)[32] and the mAP evaluation of the revised model regarding these 2 steps, pruning as well as binarization. The detailed information shown in Figure 10 tells us that after the binarization, there is a 4.36% degradation on mAP comparing with the pruned model. Comparing with benchmark YOLO v3, the model size is compressed by 97.71% at the cost of 5.32% of mAP and 9.09% of IoU.

4.2 Hardware Optimization

4.2.1 The Resource Utilization on PYNQ-Z1

As introduced in Subsection 3.3, since we binarized the hidden layer of the model, we once faced the LUT resource shortage on the PYNQ-Z1 based on the report from Vivado HLS. According to the generated report, it required 156% LUT while there was still some space for the DSP. From the report of Vivado HLS, the detail of implementation and result are represented in Figure 11.

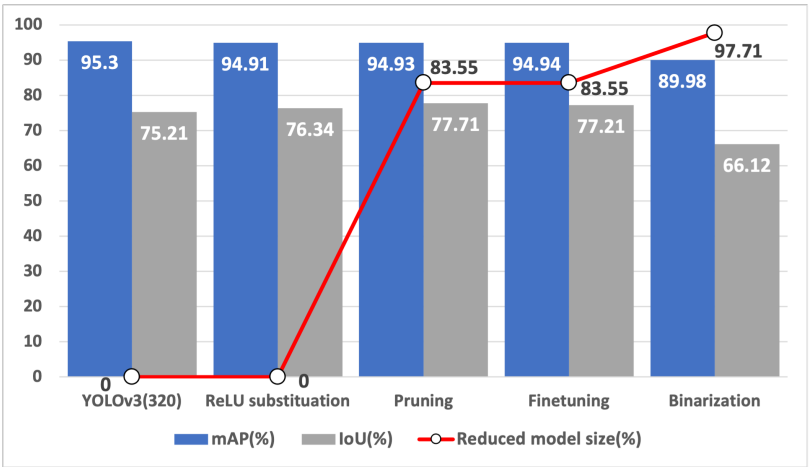


Figure 10: CNN compression evaluation

As shown in the figure, 37.2% of LUT operations in PL turn into the DSP part resulting in the increasing utilization of DSP by 15% and BRAM by 3% eventually.

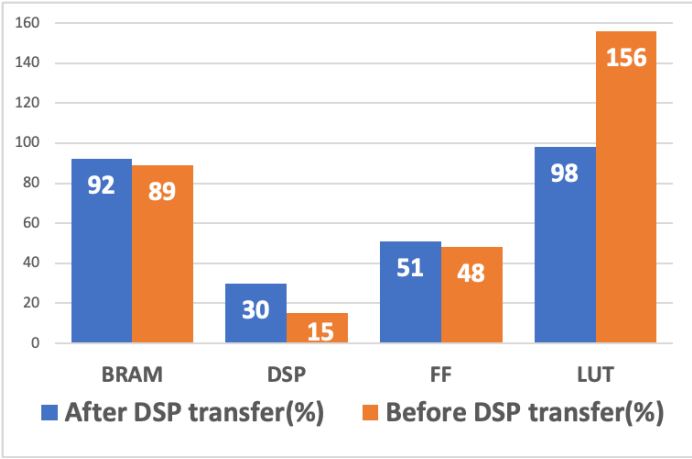


Figure 11: LUT to DSP transfer

We investigate the result of Vivado, which integrated YOLO with ZYNQ in Figure 12. BRAM takes a relatively large percentage in the whole design due to part of weight parameters need to be stored on-chip. Besides, thanks to the property of CNV in BNN, the MAC operation is replaced by pop-count and XNOR. That is the reason why the utilization rate of LUT is much higher than DSP. It is worth-mention that this system has made full use of BRAM without accessing the DDR SDRAM on the board, which ensures the acceleration of the whole system by saving the time consumption to load data from off-chip memory.

4.3 Performance and Energy Efficiency

To evaluate this methodology upon the YOLO v3 model, the performance is evaluated by the execution time and the power efficiency. We input the image with the size of 1300×1300. Regarding the time evaluation part, the model working on this PYNQ-Z1 is compared with it on ARM Cortex-A9. The data is shown in Table 6. From these data, it is obvious that the whole system on FPGA runs 22× faster than purely running it on the ARM core.

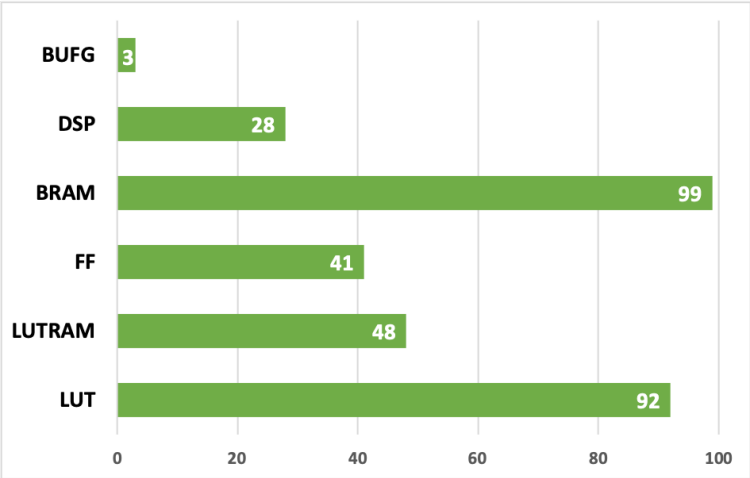


Figure 12: The resource utilization on PYNQ-Z1

Table 6: Execution time evaluation

Platform	Input layer(µs)	Hidden layer(µs)	Output layer(µs)
PYNQ-Z1	1269013	473070	63752
ARM	1269013	10264552	63752

The benefit of using FPGAs for their power efficiency. Thus, we compare our proposed design running on PYNQ-Z1 with the original design running on Intel® Xeon® CPU E5-2667. As can be observed in Table 7, although the proposed system performs not that brilliant in the performance of FPS because of the constraints of the core on the board and bounding box detection on the PS part, its power efficiency still achieves 3× better than CPU at this extreme low-cost of the platform.

Table 7: The power efficiency evaluation on the proposed system

	CPU E5-2667	PYNQ
Time (μs)	245252	2475276
Power (W)	85	2.8
FPS (sec^{-1})	4.08	0.40
Efficiency(fps/W)	0.048	0.14

4.4 The Flexibility of the Proposed Hardware Design

In this subsection, we will further verify the robustness and flexibility of our hardware design, which we explain in subsection 4.2. Besides, we will use some test cases to explain the circumstances that we can skip the step of pruning. This subsection is organized as follows, first of all, we adopt 2 other relatively thin NN models as our objects to clarify our explanation, which is demonstrated in subsection 4.4.1 and 4.4.2. Regarding these 2 models, one is the BinaryConnect topology[9] working on Cifar10, which is one of the most naive implementations of BNN proposed by Courbariaux et al., and the other one is binarized AlexNet which working on ImageNet 2012. We explain these 2 models deploying the proposed methodology in the following subsections. To simplify our explanation, we organize the construction of these two models' format as follows.

- Input = (Depth, Height, Width),

- CNV = (Depth, Kernel height, Kernel width, V_stride = 3, H_stride = 3, padding = 0),
- MaxPooling = (Kernel height, kernel width, V_stride, H_stride)

Finally, in the last subsection 4.4.3, we adapt our methodology to a more advanced FPGA board ZCU104. Regarding the NN model, we choose the binarized YOLO to compare with the result we perform on PYNQ-Z1. Besides, since the target board is more abundant in resources comparing with PYNQ-Z1, we make some optimization in HLS to relieve the burden we had on PYNQ-Z1.

4.4.1 BinaryConnect on Cifar10

To start with, we investigate the architecture of the object model. As the architecture presented in Table 9, the model consists of 5 convolutional layers to extract features in the image and 3 fully connected layers for detecting the object, which is quite similar to the VGG model[35]. It is a quite thin model, thus, we can skip the CNV Pruning and head to the Model Binarization directly as explained in subsection 4.1.1. Divergent from the original proposal of BinaryConnect to keep the full precision of the activation layer, we accept the deterministic function to binarize both weights and activation following the Algorithm 3. On the other hand, CIFAR-10 is a 10-class image classification dataset. It consists of a training set of 50000 and a test set of 10000 32×32 color images.

During training, we only binarize weights and input the image of the 32 floating points in the first layer. We use BN with a minibatch size of 50 under 100 epochs to speed up the training. We use an exponentially decaying learning rate and initialize the learning rate as 0.01, and it decays at the rate of 0.9999. It takes about 20 hours to train on Tesla P100 from scratch. Regarding the final test accuracy, the full resolution training is around 89.87% under 125 epochs, while the binarized one reaches 82.99% at the cost of 6.88% accuracy lost.

Regarding the part of the hardware optimization, we keep our hardware design. What worth mention is that different from the YOLO model using bounding boxes to detect objects, the BinaryConnect on Cifar10 utilizes the FC layer to classify objects. Thus, we can make full use of the FC-level module to perform the classification on the PL instead of the PS. Finally, we use the softmax function to give a probability distribution of the label candidates of these 10 classes. Since we don't take the softmax function implementation into our account in hardware design, this final classification is realized by the PS part using the NumPy library. For the performance evaluation, we use a size of 1300×1300 image to verify the efficiency of our hardware design. It takes 15833 microseconds to classify this image on PYNQ-Z1, while it takes 1587073 microseconds to finish all the convolution on the ARM core. Thus, we can achieve 100× acceleration through our methodology.

4.4.2 Binarized-AlexNet on ImageNet 2012

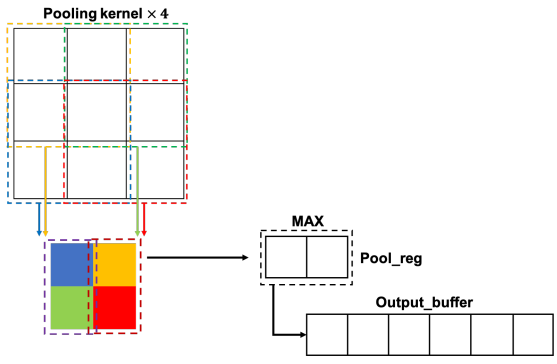


Figure 13: Pooling-level optimization for AlexNet

AlexNet[18] is a convolutional network with 8-layer deep. As shown in Table 10, it consists of 5 CNV layers, 3 max pooling layers, 2 normalization layers, and 2 FC layers. Each CNV layer consists of convolutional filters and a nonlinear AF ReLU. ImageNet 2012[10] contains 1000 categories and 1.2 million images in training data, 150,000 photographs in validation data.

Rather than a deep NN model, the model costs great amounts of memory on the parameters since the kernel sizes are 11×11, 5×5, and 3×3 instead of identified kernel size like the YOLO model. Thus, the AlexNet is much more suitable for applying weight pruning, however, not adaptable for the upcoming hardware design as discussed in subsection 2.2.1[52]. Hence, we skip the process of pruning for this model.

First of all, from the experiment of the YOLO model, we replace the ReLU function with Leaky ReLU to reduce the computational complexity. Regarding the training of the binarized AlexNet, we binarize both weight and activation. We use SGD with momentum of 0.9 for updating parameters and adopt ADAM to achieve better accuracy performance. The learning rate starts at 0.1, and it decays by 10 every 30 epochs. The BN is restrained to the mini-batch size of 256 under 100 epochs to accelerate the training.

The original model size of the full-precision AlexNet takes 475MB while the binarized one only weighs 8.2MB through the binarization. Concerning the precision performance, the binarized AlexNet achieves the Top-1 accuracy of 43.6% and Top-5 accuracy of 62.7% on ImageNet. Comparing with the original model, we achieve 98.27% of the model compression rate at the cost of 13% top-1 accuracy, 17.5% top-5 accuracy.

In the implementation of AlexNet on HLS, it is obvious that there is some change on the max pooling layer with the kernel size. Concerning the design on the pooling-level optimization introduced in subsection 3.2.3, to deal with the kernel size of 3×3, we extend the implementation of HLS design. To be more specific, as presented in Figure 13, the original 2×2 kernel design runs for 4 times to get the 3×3 pooling result. These 4 times comparisons work in inter-level pipelining with considering the resource utilization. Considering the kernel size of CNV0 and CNV1, which are 11×11 and 5×5 respectively, they can not fit the 3×3 kernel implemented in CNV-level Optimization. Thus, we offload the input layer and the first layer on the PS part and transfer the output fo CNV1 to the PL part. The rest implementations are kept as the original design, which we have introduced previously.

To evaluate the model on AlexNet, we use the same image with the size of 1300×1300. For the input layer and CNV1 convolutional layer on the PS part, it takes 3132255 microseconds. It takes 884481 microseconds for the PL part to process CNV2 CNV4 and 3 FC layers. Similar to the BinaryConnect on Cifar10, we use the softmax function to give the final probability distribution. It costs 3110680 microseconds to complete the final classification of 1000 classes. Thus, for this Alexnet model, it almost costs 7.13 seconds to classify an image of 1300×1300.

4.4.3 YOLO Implementation on ZCU104

In this subsection, we will introduce the YOLO implementation on ZCU104. Comparing with the PYNQ-Z1 board, it is a more recent device that integrates a quad-core ARM® Cortex™-A53 processing system, a dual-core Arm Cortex-R5 real-time processor. and a Mali™-400 MP2 graphics processing unit. In addition to the powerful PS part, the abundant resource on the PL part can greatly relieve the burden we have ever encountered in PYNQ-Z1. We have listed the PL specifications in Table 8.

Table 8: Key ZCU104 Specifications

Components	Size
Logic slices	504,000
6-input LUTs	230,400
LUTRAMs	101,760
Flip-Flops	460,800
Block RAM	11MB
DSP Slices	1,728

As introduced in subsection 4.2.1, on PYNQ-Z1, due to the storage of LUT resources, we once move part of logic calculations from LUT to DSP48 by returning the binarized data to 16-bit floating-point. Since the memory restriction is no longer the bottleneck for the hardware optimizaiton as Table 8 presented, there is no more necessary for us to do such a replacement. We return the data in hidden layer to 1-bit data width. On the other hand, we increase the FIFO depth through updating the **depth** option in **pragma HLS stream** without considering the resource utilization. Based on this design, we integrate the IP generated by HLS with the Zynq UltraScale PS part on Vivado.

We present the resource utilization on ZCU104 in Figure 14. As the bar graph shows, there is still a lot of space to exploit on this high-end FPGA board. Considering the purpose of this paper mainly targets the low-end FPGA board, further exploration on ZCU104 is out of the scope of our discussion. We will present the research in our future work.

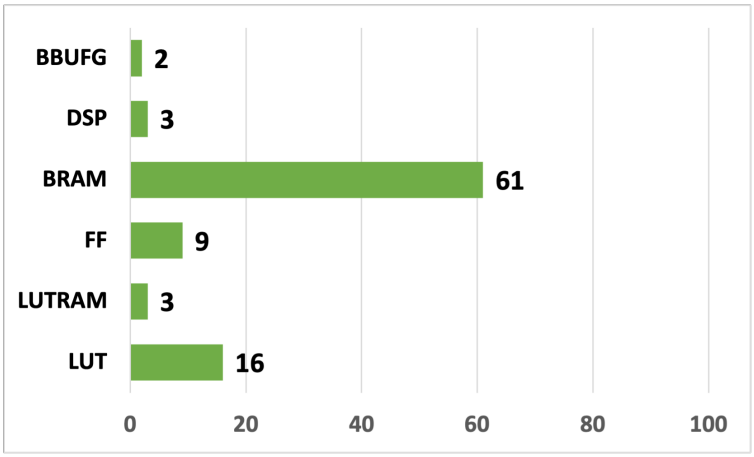


Figure 14: The resource utilization on ZCU104

Table 9: Model of BinaryConnect on Cifar10

Layer	Dimension
Input	(3, 32, 32)
CNV0	(128, 3, 3)
Batch Normalization	
Leaky ReLU	
CNV1	(128, 3, 3)
MaxPooling	(2, 2, 2, 2)
Batch Normalization	
Leaky ReLU	
CNV2	(256, 3, 3)
Batch Normalization	
Leaky ReLU	
CNV3	(256, 3, 3)
MaxPooling	(2, 2, 2, 2)
Batch Normalization	
Leaky ReLU	
CNV4	(512, 3, 3)
Batch Normalization	
Leaky ReLU	
CNV5	(512, 3, 3)
MaxPooling	(2, 2, 2, 2)
Batch Normalization	
Leaky ReLU	
FC layer	1024
Batch Normalization	
FC layer	1024
Batch Normalization	
FC layer	10

Table 10: Model of AlexNet on ImageNet

Layer	Dimension
Input	(3,227,227)
CNV0	(96, 11, 11, 4, 4)
MaxPooling	(3,3, 2, 2)
Batch Normalization	
Leaky ReLU	
CNV1	(256, 5, 5, 1, 1,2)
MaxPooling	(3, 3, 2, 2)
Batch Normalization	
Leaky ReLU	
CNV2	(384, 3, 3,1, 1, 1)
Batch Normalization	
Leaky ReLU	
CNV3	(384, 3, 3,1, 1, 1)
Batch Normalization	
Leaky ReLU	
CNV4	(256, 3, 3,1, 1, 1)
MaxPooling	(3, 3, 2, 2)
Batch Normalization	
LeakyReLU	
FC layer	4096
Batch Normalization	
FC layer	4096
Batch Normalization	
FC layer	1000

To evaluate the model performance on ZCU104, we adopt the same procedure to the system as what we did on PYNQ-Z1. The time evaluation is divided into 3 parts, the input layer working on the PS part, the hidden layer performed by the PL part of FPGA, and the final detection using the bounding box working on the PS part. As shown in Table 11, due to the powerful processing system on ZCU104, we have significantly decreased the time cost on the input and output layer. The update of FIFO depth also improves the performance on the PL part finishing the hidden layer convolution at the frequency of 107.67MHz. Through this designed system, we can achieve 10.51fps for an image size of 1300×1300.

Table 11: Execution time evaluation on ZCU104

Platform	Input layer(µs)	Hidden layer(µs)	Output layer(µs)
ZCU104	53002	29376	2750

4.4.4 Summary on the Flexibility of the Proposed Framework

To summarize, through these 2 models in subsection 4.4.1 and 4.4.2, we successfully verified the flexibility of our methodology. In dealing with some thin NN models, the pruning steps can be directly skipped and head to the model binarization. Besides, we should also admit that this methodology is not suitable for all kinds of NN models, and it is quite model-sensitive. In the experiment on the AlexNet, due to the model specificity, we have no choice to offload large amounts of data on the PS part and result in an inefficient system on PYNQ-Z1. On another note, the BinaryConnect on Cifar10 even achieves 100× acceleration comparing the model performance on the ARM core, which is mainly due to the object model makes full use of the hardware design. On the other hand, we also attempt to realize our methodology on another high-end FPGA board, ZCU104, to show the extensibility of this proposal. Finally, to review this section, we summarize the above three work's performance on the image size of 1300×1300 in Table 12.

Table 12: Performance summary on proposed methodology

Board	Model	PS run_time (µs)	PL run_time layer(µs)
PYNQ-Z1	BinaryConnect	12071	3762
PYNQ-Z1	Alexnet	6242935	884481
ZCU104	YOLO	55752	29376

4.5 Comparison with other NN FPGA Implementations

We compared our work with some other low-middle edge FPGA implementation, which includes DLAU[41] with the precision of 48-bit floating-point, 16-bit fixed-point CNP[11], FPGAConvNet[40], design for 3D CNN[37] and so on. The performance including the key features, performance (GOPS), platform as well as Power Efficiency lists in the Table 13. Although we are not comparable to [37, 53, 50, 49, 51] on their instructional performance, our work achieves a better performance in energy efficiency. Most of their work do not sacrifice their accuracy to speed up, thus, hardly can we find any useful information about the accuracy loss in their research.

Instead of only focusing on FPGA design, we downscale the design in both the algorithm level and the hardware implementation level. The methodology achieved a comparable performance. Although there is some loss in accuracy performance, the 89.98% mAP can still meet our requirement. Our instructional performance is 3 times better than that of [40] and 7 times better than that of [11]. Regarding energy efficiency, our work also achieved a satisfactory result.

Table 13: Similiar Work Comparison

Design	Platform	Testbanch	GOPS	Energy (W)	GOPS/W	Accuracy
[11]	Virtex4 SX35	Lenet-5	5.25	15	0.35	N/A
[41]	XC7Z020	DNN	N/A	N/A	N/A	negligible loss
[40]	XC7Z020	Scene Label	12.73	1.76	7.21	N/A
[37]	5SGSD8	VGG	117.8	19.1	6.17	N/A
[53]	XC7VX690T	Lenet	222.1	19.1	8.96	N/A
[50]	XC7VX485T	N/A	61.12	18.61	3.31	N/A
[49]	Stratix V	AlexNet	123.5	13.18	9.37	N/A
[51]	XC7Z020+ XC7VX690T×6	AlexNet	1280.3	160	8.00	N/A
[6]	XC7VX485T	LSTM-RNN	7.26	19.63	0.37	N/A
Our work	PYNQ-Z1	Tiny YOLO-v3	38.1	2.8	13.61	-5.32%

5 Conclusion

We proposed a methodology for a NN implementation on an extremely low-end board. Regarding the case study of YOLO v3, the network architecture is downscaled by 97.71% at the sacrifice of the accuracy of about 5.32%. In the part of hardware design, through the realization of the pipelining, loop unrolling, and so on, the FPGA part is accelerated about 22× faster than it purely running on ARM core and realizes 3× power efficiency than the original tiny YOLO on CPU. Besides, to verify the flexibility of this methodology, we successfully adapt 2 other models, BinaryConnect on Cifar10 and Binarized Alexnet on ImageNet, to PYNQ-Z1 through this methodology. Model of BinaryConnect on Cifar10 even achieves around 100× acceleration comparing with it purely running on the PYNQ-Z1 ARM core. On the other hand, we further embed the binarized YOLO model on a high-end FPGA target ZCU104 and achieve 10.51 fps on an image size of 1300×1300

In the future, we will further improve the present work by deploying the algorithm of memory compression to strive for more space to improve system performance. Besides, a Quantized Neural Network (QNN) will also be challenged in this methodology to evaluate these two light-weight algorithms' performance on this system in a quantitative way. As revealed in subsection 4.4.3, there is still a lot of space to exploit on ZCU104. A further discussion on this board is also the key to improve this work.

References

[1] A. Ansari and T. Ogunfunmi. Elasto-net: An hdl conversion framework for convolutional neural networks. In *2018 52nd Asilomar Conference on Signals, Systems, and Computers*, pages 787–791, 2018.

[2] Steven M. Beitzel, Eric C. Jensen, and Ophir Frieder. *MAP*, pages 1691–1692. Springer US, Boston, MA, 2009.

[3] Michaela Blott, Thomas B Preußer, Nicholas J Fraser, Giulio Gambardella, Kenneth O'brien, Yaman Umuroglu, Miriam Leeser, and Kees Vissers. Finn-r: An end-to-end deep-learning framework for fast exploration of quantized neural networks. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 11(3):1–23, 2018.

[4] Charles Boncelet, George Bockari, Unknown, Luis Rodrígues-Flores, Nhan Truong, A. Luna, Tim, Muhammed Abdelshakour, and Dan Buskirk. Pynq-z1: Python productivity for zynq-7000 arm/fpga soc.

[5] A. Bulat, J. Kossaifi, G. Tzimiropoulos, and M. Pantic. Toward fast and accurate human pose estimation via soft-gated skip connections. In *2020 15th IEEE International Conference on Automatic Face and Gesture Recognition (FG 2020)*, pages 8–15, 2020.

[6] Andre Xian Ming Chang, Berin Martini, and Eugenio Culurciello. Recurrent neural networks hardware implementation on fpga, 2016.

[7] Matthieu Courbariaux and Yoshua Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, abs/1602.02830, 2016.

[8] Matthieu Courbariaux and Yoshua Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, abs/1602.02830, 2016.

[9] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'15, page 3123–3131, Cambridge, MA, USA, 2015. MIT Press.

[10] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[11] 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*, pages 32–37, 2009.

[12] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, and H. Yang. Angel-eye: A complete design flow for mapping cnn onto embedded fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(1):35–47, 2018.

[13] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, and H. Yang. Angel-eye: A complete design flow for mapping cnn onto embedded fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(1):35–47, 2018.

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[15] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger. Densely connected convolutional networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2261–2269, 2017.

[16] Eldar Insafutdinov, Leonid Pishchulin, Bjoern Andres, Mykhaylo Andriluka, and Bernt Schiele. Deepercut: A deeper, stronger, and faster multi-person pose estimation model. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Computer Vision – ECCV 2016*, pages 34–50, Cham, 2016. Springer International Publishing.

[17] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, page 1097–1105, Red Hook, NY, USA, 2012. Curran Associates Inc.

[19] K.Wei, K.Honda, and H.Amano. An implementation methodology for neural network on a low-end fpga board. In *2020 The Eighth International Symposium on Computing and Networking (CANDAR)*, 11 2020.

[20] Chen-Yu Lee, Patrick W. Gallagher, and Zhuowen Tu. Generalizing pooling functions in convolutional neural networks: Mixed, gated, and tree, 2015.

[21] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014.

[22] Zechun Liu, Baoyuan Wu, Wenhan Luo, Xin Yang, Wei Liu, and Kwang-Ting Cheng. Bi-real net: Enhancing the performance of 1-bit cnns with improved representational capability and advanced training algorithm. *CoRR*, abs/1808.00278, 2018.

[23] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[24] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. *CoRR*, abs/1411.4038, 2014.

[25] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, page 45–54, New York, NY, USA, 2017. Association for Computing Machinery.

[26] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, page 45–54, New York, NY, USA, 2017. Association for Computing Machinery.

[27] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation functions: Comparison of trends in practice and research for deep learning. *CoRR*, abs/1811.03378, 2018.

[28] University of Washington. YOLO: Real-Time Object Detection, July 2016. Available online at `https://pjreddie.com/darknet/yolo/`.

[29] Hieu Pham, Zihang Dai, Qizhe Xie, Minh-Thang Luong, and Quoc V. Le. Meta pseudo labels, 2021.

[30] L. Ranganath, D. J. Kumar, and P. S. N. Reddy. Design of mac unit in artificial neural network architecture using verilog hdl. In *2016 International Conference on Signal Processing, Communication, Power and Embedded System (SCOPES)*, pages 607–612, 2016.

[31] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. *CoRR*, abs/1603.05279, 2016.

[32] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *CoRR*, abs/1804.02767, 2018.

[33] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, page 91–99, Cambridge, MA, USA, 2015. MIT Press.

[34] Suhap Sahin, Yasar Becerikli, and Suleyman Yazici. Neural network implementation in hardware using fpgas. In Irwin King, Jun Wang, Lai-Wan Chan, and DeLiang Wang, editors, *Neural Information Processing*, pages 1105–1112, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[35] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015.

[36] Zhihui Su, Ming Ye, Guohui Zhang, Lei Dai, and Jianda Sheng. Improvement multi-stage model for human pose estimation. *CoRR*, abs/1902.07837, 2019.

[37] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, page 16–25, New York, NY, USA, 2016. Association for Computing Machinery.

[38] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014.

[39] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, pages 65–74. ACM, 2017.

[40] S. I. Venieris and C. Bouganis. fpgaconvnet: A framework for mapping convolutional neural networks on fpgas. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 40–47, 2016.

[41] C. Wang, L. Gong, Q. Yu, X. Li, Y. Xie, and X. Zhou. Dlau: A scalable deep learning accelerator unit on fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(3):513–517, 2017.

[42] Chien-Yao Wang, Alexey Bochkovskiy, and Hong-Yuan Mark Liao. Scaled-yolov4: Scaling cross stage partial network, 2020.

[43] Erwei Wang, James J. Davis, Peter Y. K. Cheung, and George A. Constantinides. Lutnet: Rethinking inference in FPGA soft logic. *CoRR*, abs/1904.00938, 2019.

[44] Y. Wang, J. Xu, Y. Han, H. Li, and X. Li. Deepburning: Automatic generation of fpga-based learning accelerators for the neural network family. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2016.

[45] Bin Yang, Lin Yang, Xiaochun Li, Wenhan Zhang, Hua Zhou, Yequn Zhang, Yongxiong Ren, and Yinbo Shi. 2-bit model compression of deep convolutional neural network on ASIC engine for image retrieval. *CoRR*, abs/1905.03362, 2019.

[46] Tao Yang, Yan Wu, Junqiao Zhao, and Linting Guan. Semantic segmentation via highly fused convolutional network with multiple soft cost functions. *CoRR*, abs/1801.01317, 2018.

[47] Tao Yang, Yan Wu, Junqiao Zhao, and Linting Guan. Semantic segmentation via highly fused convolutional network with multiple soft cost functions. *CoRR*, abs/1801.01317, 2018.

[48] Yufei Ma, N. Suda, Yu Cao, J. Seo, and S. Vrudhula. Scalable and modularized rtl compilation of convolutional neural networks onto fpga. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, 2016.

[49] C. Zhang and V. Prasanna. Frequency domain acceleration of convolutional neural networks on cpu-fpga shared memory system. *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017.

[50] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason 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*, FPGA '15, page 161–170, New York, NY, USA, 2015. Association for Computing Machinery.

[51] Chen Zhang, Di Wu, Jiayu Sun, Guangyu Sun, Guojie Luo, and Jason Cong. Energy-efficient cnn implementation on a deeply pipelined fpga cluster. In *ISLPED*, pages 326–331. ACM, 2016.

[52] Min Zhang, Linpeng Li, Hai Wang, Yan Liu, Hongbo Qin, and Wei Zhao. Optimized compression for implementing convolutional neural networks on fpga. *Electronics*, 8:295, 03 2019.

[53] Zhiqiang Liu, Yong Dou, Jingfei Jiang, and Jinwei Xu. Automatic code generation of convolutional neural networks in fpga implementation. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 61–68, 2016.

[54] Shuchang Zhou, Zekun Ni, Xinyu Zhou, He Wen, Yuxin Wu, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *CoRR*, abs/1606.06160, 2016.

[55] Barret Zoph, Golnaz Ghiasi, Tsung-Yi Lin, Yin Cui, Hanxiao Liu, Ekin D. Cubuk, and Quoc V. Le. Rethinking pre-training and self-training, 2020.