REQ-YOLO: A Resource-Aware, Efficient Quantization Framework for Object Detection on FPGAs

Caiwen Ding[2,+], Shuo Wang[1,+], Ning Liu[2], Kaidi Xu[2], Yanzhi Wang[2] and Yun Liang[1,3,*]
[+]These authors contributed equally.
[1]Center for Energy-Efficient Computing & Applications (CECA), School of EECS, Peking University, China
[2]Department of Electrical & Computer Engineering, Northeastern University, Boston, MA, USA
[3]Peng Cheng Laboratory, Shenzhen, China
[1]{shvowang, ericlyun}@pku.edu.cn, [2]{ding.ca, liu.ning, xu.kaid}@husky.neu.edu, [2]yanz.wang@northeastern.edu

ABSTRACT

Deep neural networks (DNNs), as the basis of object detection, will play a key role in the development of future autonomous systems with full autonomy. The autonomous systems have special requirements of real-time, energy-efficient implementations of DNNs on a power-constrained system. Two research thrusts are dedicated to performance and energy efficiency enhancement of the inference phase of DNNs. The first one is model compression techniques while the second is efficient hardware implementation. Recent works on extremely-low-bit CNNs such as the binary neural network (BNN) and XNOR-Net replace the traditional floating point operations with binary bit operations which significantly reduces the memory bandwidth and storage requirement. However, it suffers from non-negligible accuracy loss and underutilized digital signal processing (DSP) blocks of FPGAs.

To overcome these limitations, this paper proposes REQ-YOLO, a resource aware, systematic weight quantization framework for object detection, considering both algorithm and hardware resource aspects in object detection. We adopt the block-circulant matrix method and propose a heterogeneous weight quantization using *Alternative Direction Method of Multipliers* (ADMM), an effective optimization technique for general, non-convex optimization problems. To achieve real-time, highly-efficient implementations on FPGA, we present the detailed hardware implementation of block circulant matrices on CONV layers and develop an efficient processing element (PE) structure supporting the heterogeneous weight quantization, CONV dataflow and pipelining techniques, design optimization, and a template-based automatic synthesis framework to optimally exploit hardware resource. Experimental results show that our proposed REQ-YOLO framework can significantly compress the YOLO model while introducing very small accuracy degradation. The related codes are here: https://github.com/Anonymous788/heterogeneous_ADMM_YOLO.

*Corresponding author.

FPGA '19, February 24–26, 2019, Seaside, CA, USA

ACM ISBN 978-1-4503-6137-8/19/02...$15.00
https://doi.org/10.1145/3289602.3293904

KEYWORDS

FPGA; YOLO; object detection; compression; ADMM

ACM Reference Format:
Caiwen Ding, Shuo Wang, Ning Liu, Kaidi Xu, Yanzhi Wang and Yun Liang. 2019. REQ-YOLO: A Resource-Aware, Efficient Quantization Framework for Object Detection on FPGAs. In *The 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '19), Feb. 24–26, 2019, Seaside, CA, USA.* ACM, NY, NY. 10 pages. DOI: https://doi.org/10.1145/3289602.3293904

1 INTRODUCTION

Autonomous systems such as unmanned aerial vehicles (UAVs), autonomous underwater vehicles (AUVs), and unmanned ground vehicles (UGVs) have been rapidly growing for performing surveillance, object detection [2], and object delivery [21] tasks in scientific, military, agricultural, and commercial applications. The full autonomy of such systems relies on the integration of artificial intelligence software with hardware.

The deployment of deep neural networks (DNNs) in autonomous systems include multiple aspects, i.e., object detection/surveillance algorithms, and advanced control (e.g., deep reinforcement learning technique). Since DNN-based advanced control is not widely in place yet, we focus on the former aspect. Object detection algorithms are different from image classification [26] in that the former need to simultaneously detect and track multiple objects with different sizes. Representative object detection algorithms include R-CNN [14] and YOLO [38]. The autonomous system applications have special requirements of real-time, energy-efficient implementations on a power-constrained system.

Two research thrusts are dedicated to performance and energy efficiency enhancement of the inference phase of DNNs. The first one is model compression techniques for DNNs [16, 29, 52, 9], including weight pruning, weight quantization, low-rank approximation, etc. S. Han et al. [16] have proposed an iterative DNN weight pruning method, which could achieve 9× weight reduction on the AlexNet model and has been applied to LSTM RNN as well [18]. However, this method results in irregularity in weight storage, and thereby degrades the parallelism degree and hardware performance, as observed in [46, 10, 51]. Recent work [10, 46] adopts block-circulant matrices for weight representation in DNNs in both image classification DNN [10] and LSTM RNN [46] tasks. This method is demonstrated to achieve higher hardware performance than iterative pruning due to the regularity in weight storage and computation. The second one is efficient hardware implementations, including FPGAs and ASICs [50, 7, 54, 36, 1, 31, 32, 49, 53]. FPGAs are gaining more popularity for striking a balance

between high hardware performance and fast development round. A customized hardware solution on FPGA can offer significant improvements in energy efficiency and power consumption compared to CPU and GPU clusters.

Convolutional (CONV) layers are more computation-intensive than fully-connected (FC) layers. Recently CONV layers are becoming more important in state-of-the-art DNNs [39, 26]. Extremely-low-bit CNNs such as the binary neural network (BNN) [9] and XNOR-Net[37] have demonstrated hardware friendly ability on FPGAs [45]. Binarization not only reduces memory bandwidth and storage requirement but also replaces the traditional floating point operations with binary bit operations, which can be efficiently implemented on the look-up-tables (LUT)-based FPGA chip, whereas suffering non-negligible accuracy degradation on large datasets due to the over-quantized weight representation. More importantly, the majority of DSP resource will be wasted due to the replacement of multipliers, introducing significant overhead on LUTs. Overall, there lacks a systematic weight quantization framework considering hardware resource aspect on FPGAs. In addition, despite the research efforts devoted to the hardware implementation of image classification tasks [26, 27, 19], there lacks enough investigation on the hardware acceleration of object detection tasks.

In this paper, we propose REQ-YOLO, a resource-aware, efficient weight quantization framework for object detection by exploring both software and hardware-level optimization opportunities on FPGAs. We adopt the block-circulant matrix based compression technique and propose a heterogeneous weight quantization using ADMM on the FFT results considering hardware resource. It is necessary to note that the proposed framework is also applicable to other model compression techniques. To enable real-time, highly-efficient implementations on FPGA, we present the detailed hardware implementation of block circulant matrices on CONV layers and develop an efficient processing element (PE) structure supporting the heterogeneous weight quantization method, dataflow based pipelining, design optimization, and a template-based automatic synthesis framework.

Our specific contributions are as follows:

- We present a detailed hardware implementation and optimization of block circulant matrices on CONV layers on object detection tasks.
- We present a heterogeneous weight quantization method including both equal-distance and mixed powers-of-two methods considering hardware resource on FPGAs. We adopt ADMM to directly quantize the FFT results of weight.
- We employ an HLS design methodology for productive development and optimal hardware resource exploration of our FPGA-based YOLO accelerator.

Experimental results show that our proposed REQ-YOLO framework can significantly compress the YOLO model while introducing very small accuracy degradation. Our framework is very suitable for FPGA and the associated YOLO implementations outperform the state-of-the-art designs on FPGAs.

2 PRELIMINARIES ON OBJECT DETECTION

Deep neural networks (DNNs) have dominated the state-of-the-art techniques of object detection. There are typically two main types of object detection methods: (i) region proposal based method and (ii) proposal-free method. For the region proposal based methods,

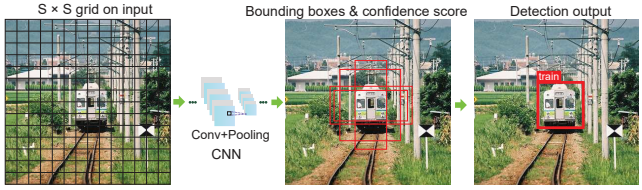

Figure 1: Example of object dection using YOLO.

R-CNN first generates potential object regions and then performs classification on the proposed regions [12]. SPPnet [20], Fast R-CNN [13], and Faster R-CNN [41] are typical in this category. As for the proposal-free methods, MS-CNN [5] proposes a unified multi-scale CNN for fast object detection. YOLO [38] simultaneously predicts multiple bounding boxes and classification class probabilities. Compared to the region proposal-based methods, YOLO does not require a second classification operation for each region and therefore it achieves significant faster speed. However, YOLO suffers from several drawbacks: (i) YOLO makes a significant number of localization errors compared to Fast R-CNN; (ii) Compared to region proposal-based methods, YOLO has a relatively low recall. To improve the localization and recall while maintaining classification accuracy, YOLO v2 [39] has been proposed. In this paper, we focus on an embedded version of YOLO - tiny YOLO [44] for hardware implementation. Compared to other versions such as YOLO v2 [39], v3 [40], and YOLO [38], it has a smaller network structure and fewer weight parameters, but with tolerable accuracy degradation.

2.1 You Only Look Once (YOLO) Network

Fig. 1 shows an overview of object detection and tiny YOLO application. It uses CONV layers to extract features from images, anchor boxes to predict bounding boxes, and regression for object detection, based on the yolov2 tiny [44] framework. The input image (frame) is separated into an $S \times S$ grid. Each grid cell detects an object and predicts B bounding boxes and the corresponding confidence scores when the grid cell and the center of object overlap each other. Typically $S = 13$ and $B = 5$. For each bounding box, there are 5 predictions made: x, y, w, h, and a confidence score. (x, y) is the coordinates of the box center located in the grid cell, and w and h are the width and height of the bounding box. The confidence score is defined as $Pr(Obj) \times IOU_{pred}^{truth}$, where $Pr(Obj)$ is the prediction probability and IOU_{pred}^{truth} is the *intersection over union* (IOU). Here, IOU is determined by dividing the area of overlap between the predicted bounding box and its corresponding ground-truth bounding box by the area of union.

There are C *conditional class probabilities* on the grid cell containing the object, $Pr(Class_i|Obj)$, predicted by each grid cell. The *class-specific confidence scores* for each box are calculated as follows:

$$Pr(Class_i|Obj) \cdot Pr(Obj) \cdot IOU_{pred}^{truth} = Pr(Class_i) \cdot IOU_{pred}^{truth} \quad (1)$$

The scores represent how accurate the box is pertinent to the object and the probability of the object class. These predictions are encoded as an $S \times S \times (B \cdot 5 + C)$) tensor. Fig. 2 shows the architecture of tiny YOLO. It has 9 CONV layers. The input images are re-sized to 416 by 416 from the PASCAL 2007 detection dataset [12]. From the 1^{st} to the 8^{th} layer, a 3×3 CONV operation (stride 1 and zero padding with 1) is followed by a max pooling operation with 2×2

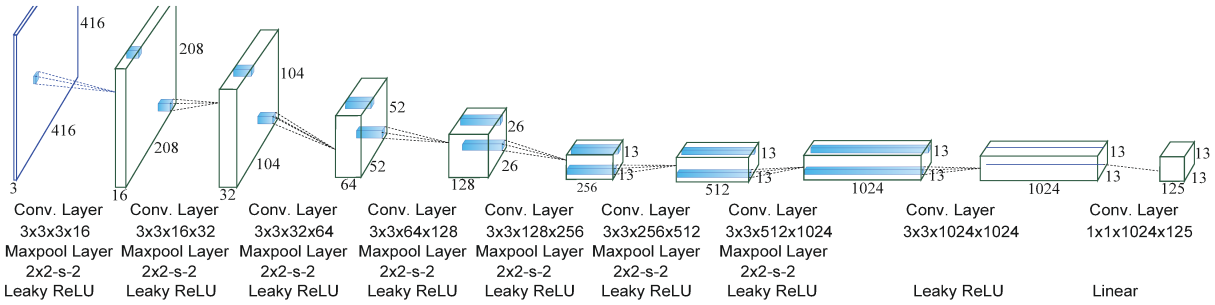

Figure 2: The tiny YOLO architecture with convolution layers.

filters and stride 2. In the last CONV layer, the 1×1 CONV operation reduces the feature space from the previous layer.

2.2 Convolutional (CONV) Layers

Convolutional (CONV) layers convolve each input feature map with an $r \times r$ weight filter. The convolutional results are then accumulated, added with a bias. After passing the intermediate result through a activation function (such as rectified linear unit (ReLU), sigmoid, or hyperbolic tangent (tanh)), we produce a single output feature map. Repeat this procedure for the rest of weight filters, we obtain the whole output feature map. Zero padding is adopted at the border to ensure the output feature maps have the same size as input. The expression of the CONV layer operation is shown in Equation (2).

$$\mathbf{y}_{c'} = f\Big(\sum_{c=1}^{C} \mathbf{x}_c * \mathbf{w}_{c,c'} + \mathbf{b}_{c'}\Big) \tag{2}$$

there are C input feature maps $(\mathbf{x}_1; \mathbf{x}_2; \mathbf{x}_3; ...; \mathbf{x}_C)$ and C' output feature maps $(\mathbf{y}_1; \mathbf{y}_2; \mathbf{y}_3; ...; \mathbf{y}_{C'})$, respectively. The weight parameters of this CONV layer $\mathbf{W}$ has the shape of $[filter\_height,\ filter\_width,\ in\_channels,\ out\_channels]$, denoted as $\mathbf{W} \in \mathbb{R}^{r \times r \times C \times C'}$. f is the activation function and $\mathbf{b}$ is the bias as the same size as output feature maps.

2.3 Pooling layer and other types of layers

In YOLO network structure, pooling layers downsample each input feature map by passing it through a 2×2 max window (pool) with a stride of 2, resulting in no overlapped regions. Pooling layers reduce the data dimensions and mitigate overfitting issues. Max pooling is the dominant type of pooling strategy in state-of-the-art DCNNs due to its higher overall accuracy and convergence speed [7]. Batch normalization (BN) normalizes the variance and mean of the features across examples in each mini-batch [22], to avoid the gradient vanishing or explosion problems [23].

3 COMPRESSED CONVOLUTION LAYERS

3.1 Block-Circulant Matrices

We can reducing weight storage by replacing the original weight matrix with one or multiple blocks of circulant matrices, where each row/column is the cyclic reformulation of the others, as shown in Equantion (3).

$$\mathbf{W}_{ij} = \begin{bmatrix} w_1 & w_{L_b} & \cdots & w_3 & w_2 \\ w_2 & w_1 & \cdots & w_4 & w_3 \\ \vdots & \vdots & \ddots & \vdots & \\ w_{L_{b-1}} & w_{L_{b-2}} & \cdots & w_1 & w_{L_b} \\ w_{L_b} & w_{L_{b-1}} & \cdots & w_2 & w_1 \end{bmatrix}, \tag{3}$$

where L_b represents the row/column size of each structured matrix (or block size, FFT size).

3.2 Block-Circulant Matrices-Based CONV

CirCNN [10] and C-LSTM [46] have a detailed discussion of the inference algorithm for block circulant matrix-based DNNs. The theoretical foundation is derived in [55], which shows that the "effectiveness" of block circulant matrix-based DNNs is the same with DNNs without compression. However, CirCNN and C-LSTM do not thoroughly discuss convolutional (CONV) layers which are the major computation in state-of-the-art DNNs. In this section, we present the detailed formulation of block circulant matrix-based computation for CONV layers, which are the major computation part of the tiny YOLO algorithm.

Given an input and weight filters (tensor), in a 2-D convolution operation, we slide each filter over all spatial locations of the input, multiply the corresponding entries of the input and filter, and accumulate the intermediate product values. The result is the output of the 2-D convolution. It is well-known that the multiplication-and-accumulation (MAC) operation dominates the overall convolution computation, and will be our focus of acceleration.

Each 4-D weight tensor $\mathbf{W}$ has the shape of $\mathbf{W} \in \mathbb{R}^{r \times r \times C \times C'}$. Using block circulant matrix, we compress the input channels C and output channels C' plane of the 4-D weight tensor. According to the *circulant convolution theorem* [35, 43], instead of directly performing the matrix-vector multiplication, we could use the FFT-based fast multiplication method. In each block circulant matrix, only the first row is needed for calculation, and is termed the *index vector*. The calculation of a block circulant matrix-vector multiplication $\mathbf{W}_{ij}\mathbf{x}_j$ can be performed as follows.

$$\mathbf{a} = \mathbf{W}_{ij}\mathbf{x}_j = \mathbf{w}_{ij} \circledast \mathbf{x}_j = \text{IFFT}\Big(\text{FFT}(\mathbf{w}_{ij}) \circ \text{FFT}(\mathbf{x}_j)\Big) \tag{4}$$

where '$\circledast$' denotes circular convolution, and $\circ$ represents element-wise multiplication. After the weight compression, the shape of the weight tensor becomes $r \times r \times C \times C'/L_b$. For better illustration, we stretch the compressed weight tensor from the 2-D $r \times r$ matrix to a 1-D r^2 vector. This procedure is shown in Fig. 3 (a). The outer-level brackets indicate the IDs of the output channels for weight tensor (i.e., 1, 2, ..., C'/L_b). The inner-level brackets show the IDs of index vectors for input channels (i.e., 1, 2, ..., C/L_b), where each vector with length L_b corresponds to a block circulant matrix.

Fig. 3 (b) illustrates the block circulant matrix-based CONV operation. We slide each block of FFT results of the weight kernel $\text{FFT}(\mathbf{w}_{ij})$ (marked as blue bars) over all spatial locations of the input feature maps $\text{FFT}(\mathbf{x}_j)$ of FFT results with zero padding (marked as white bars and dotted lines). We then multiply the corresponding

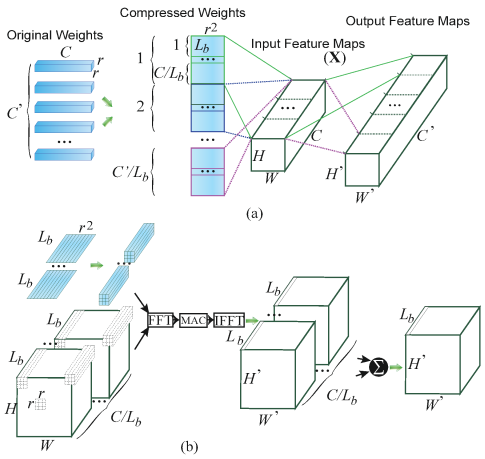

Figure 3: A block circulant matrix-based CONV layer.

FFT results of the input feature map $\text{FFT}(\mathbf{x}_j)$ and weight kernel $\text{FFT}(\mathbf{w}_{ij})$. The accumulation of the r^2 multiplication results will be sent to IFFT computing module and become the output of the block circulant matrix-based CONV operation. The forward propagation process in the CONV layer is shown in Fig. 3 (a). Please note that each weight kernel has C/L_b blocks. After finding the CONV kernel result, we sum up all the intermediate matrices and output a single matrix. This becomes one channel at the output of CONV layer. In other words, we use the first C-channel of the weight kernel to compute the first output channel of CONV layer, and so on. In addition, each output channel has its own batch normalization and bias, which will be calculated in the end.

4 THE REQ-YOLO FRAMEWORK

4.1 Heterogeneous Weight Quantization

The previous works on weight quantization [30, 52] has demonstrated the effectiveness of various quantization techniques applied to DNN models including fixed bit-length, ternary and even binary weight representations. Weight quantization can simultaneously reduce the DNN model size, computation and memory access intensity [9]. Some the other prior works have investigated the combination of equal-distance weight quantization and other mode compression techniques such as weight pruning [17, 18].

Equal-distance quantization [9], to some extent, facilitates efficient hardware implementations while maintaining accuracy requirement, whereas the power consumption and hardware resource utilization of the involved multiplications is still high. On the other hand, the powers-of-two quantization technique is extremely hardware efficient by using binary bit shift-based multiplication, however, suffering non-negligible accuracy degradation due to the highly unevenly spaced scales. To overcome the accuracy degradation problem and maintain low power consumption and resource utilization, we propose a heterogeneous weight quantization technique, i.e., we (i) adopt the equal-distance quantization for some CONV layers and (ii) we use the mixed powers-of-two-based quantization for other CONV layers. Please note that the quantization technique is identical inside each CONV layer. The mixed powers-of-two-based weight representation consists of a sign bit part and a magnitude bits part. The magnitude bits part is the combination

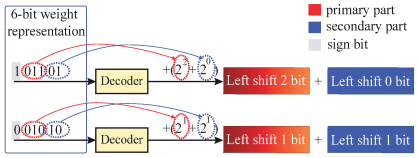

Figure 4: An illustration of the 6-bit weight representation using the mixed powers-of-two quantization.

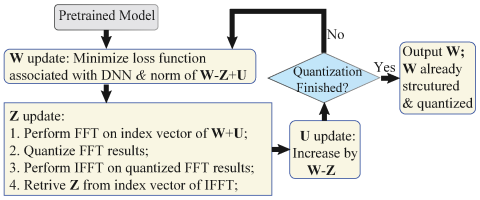

Figure 5: The overall procedure of ADMM-based weight quantization on FFT results.

of a primary powers-of-two and a secondary powers-of-two part. It not only enhances the model accuracy by mitigating the uneven data scaling problem but also facilitates efficient hardware implementations. Fig. 4 illustrates a 6-bit weight representation using the mixed powers-of-two quantization method. 1 bit is for representing the sign bit, and 5-bit are for magnitude bits (in which the first 3-bit are primary; the last 2-bit are secondary). The primary and secondary part of the weight "101101" are decoded as "011" and "01", respectively. Therefore, when multiplying an input value by weight "101101", we shift the input left 2 bit and 0 bit, respectively, and sum the two products up.

4.2 ADMM for Weight Quantization

In the hardware implementation, for each block circulant matrix $\mathbf{W}_{ij}$, we actually store the FFT result $\text{FFT}(\mathbf{w}_{ij})$ instead of the index vector $\mathbf{w}_{ij}$ [10]. However, it is not straightforward to directly apply quantization on the FFT results $\text{FFT}(\mathbf{w}_{ij})$'s because of the difficulty of impact evaluations. This is a major limitation of the prior work [17, 18, 10, 46], which would be further exacerbated because both real and imaginary parts of FFT results need to be stored.

To overcome this limitation, in this section we incorporate ADMM with FFT/IFFT and use it for the heterogeneous weight quantization to directly quantize the FFT results $\text{FFT}(\mathbf{w}_{ij})$'s, which can achieve higher compression ratio and lower accuracy degradation compared with prior works. This novel method effectively leverages the flexibility in ADMM. In a nutshell, we propose to *perform quantization in the frequency (FFT) domain and perform weight mapping in the weight domain.* Details are described as follows, as also shown in Fig. 5.

Consider the quantization problem as an optimization problem $\min_{\mathbf{x}} f(\mathbf{x})$ with combinatorial constraints. This problem is difficult to solve directly using optimization tools. Through the application of ADMM [4, 24], the original quantization problem is decomposed into two subproblems, which will be iteratively solved until convergence. The first subproblem is $\min_{\mathbf{x}} f(\mathbf{x}) + q_1(\mathbf{x})$ where $q_1(\mathbf{x})$ is a differentiatede, quadratic term. This subproblem does not

have combinatorial constraints and can be solved using traditional optimization method, e.g., stochastic gradient descent for DNN training. The second subproblem is: $\min_{\mathbf{x}} g(\mathbf{x}) + q_2(\mathbf{x})$, where $g(\mathbf{x})$ corresponds to the original combinatorial constraints and $q_2(\mathbf{x})$ is another quadratic term. For special types of combinatorial constraints, including block circulant matrices, quantization, etc., the second subproblem can be optimally and analytically solved, as we will see in the following discussions.

In the tiny YOLO network, the weights in the l^{th} layer is denoted by $\mathbf{W}_l$. The loss function is represented by $f\left(\{\mathbf{W}_l\}_{l=1}^N\right)$. Assume a weight sub-matrix $(\mathbf{W}_l)_{ij} \in \mathbb{R}^{L_b \times L_b}$ is mapped to a block circulant matrix. Directly training the network in the structured format will incur a large number of equality constraints (to maintain the structure). This makes the training problem inefficient to solve using the conventional stochastic gradient descent. On the other hand, ADMM can be utilized to efficiently solve this problem, and a large number of equality constraints can be avoided.

We introduce auxiliary variables $\mathbf{Z}_l$ and $\mathbf{U}_l$, which have the same dimensionality as $\mathbf{W}_l$. Through the application of ADMM[1], the original structured training problem can be decomposed into two subproblems, which are iteratively solved until convergence. In each iteration k, the first subproblem is

$$\underset{\{\mathbf{W}_l\}}{\text{minimize}} \quad f\left(\{\mathbf{W}_l\}_{l=1}^N\right) + \sum_{l=1}^{N} \frac{\rho_l}{2} \|\mathbf{W}_l - \mathbf{Z}_l^k + \mathbf{U}_l^k\|_F^2, \tag{5}$$

where $\mathbf{U}_l^k$ is the dual variable updated in each iteration, $\mathbf{U}_l^k := \mathbf{U}_l^{k-1} + \mathbf{W}_l^k - \mathbf{Z}_l^k$. In the objective function of (5), the first term is the differentiable loss function, and the second quadratic term is differentiable and convex. Thus, this subproblem can be solved by stochastic gradient descent and the complexity is the same as training the original DNN. A large number of constraints are avoided here. The result of the first subproblem is denoted by $\mathbf{W}_l^{k+1}$.

The second subproblem, on the other hand, is to quantize $\mathbf{W}_l^{k+1} + \mathbf{U}_l^k$ in the frequency domain, and the result of the second subproblem is denoted by $\mathbf{Z}_l^{k+1}$. For a matrix $(\mathbf{W}_l^{k+1} + \mathbf{U}_l^k)_{ij}$ for frequency-domain quantization, we first perform FFT on the index vector. Then we perform quantization on the FFT results. For the equal-distance quantization, we constrain them on a set of quantization levels $\alpha \times \{-(\frac{M}{2}-1), ..., -1, 0, 1, 2, ..., \frac{M}{2}-1\}$ associated with a layerwise coefficient α, where M is the predefined number of quantization levels; for the mixed powers-of-two quantization, we constraint them to $\alpha \times \{0, \pm 2^0, \pm 2^1, \pm 2^2, ..., \pm 2^{M_1}\} \bigcup \{0, \pm 2^0, \pm 2^1, \pm 2^2, ..., \pm 2^{M_2}\}$, where M_1 and M_2 is the total number of bits in the primary and secondary part, respectively.

This step is simply mapping each FFT value to the nearest quantization level. The quantization levels are determined by the (i) range of FFT results of index vector, and (ii) the predefined number S of quantization levels. The coefficient α may be different for different layers, which will not increase hardware implementation complexity because α will be stored along with the FFT results after quantization. Finally, as the key step, we perform IFFT on the quantized FFT results, and the restored vector becomes the index vector of $(\mathbf{Z}_l^{k+1})_{ij}$. We then retrieve block-circulant matrix $\mathbf{Z}_l^{k+1}$ from the index vector after IFFT.

[1]The details of the ADMM algorithm are discussed in [4]. We omit the details because of space limitation.

We have proved that the above frequency-domain quantization procedure is the optimal, analytical solution of the second subproblem. Because of the symmetric property of quantization above and below 0, the restored index vector of $(\mathbf{Z}_l^{k+1})_{ij}$ will still be a real-valued vector. Besides, the frequency-domain quantization procedure is applied after the block circulant matrix training of DNNs, and the circulant structure will be maintained in quantization. This is because we restore a single index vector for $(\mathbf{Z}_l^{k+1})_{ij}$ and thus $(\mathbf{Z}_l^{k+1})_{ij}$ will maintain the imposed circulant structure. After the convergence of ADMM, the solution $\mathbf{W}_l$ meets the two requirements: (i) the block-circulant structure, and (ii) the FFT results are quantized.

5 HARDWARE IMPLEMENTATION

In this section, we implement the YOLO-based object detection on FPGAs. In order to achieve both low-power and high-performance, the proposed REQ-YOLO framework ensures that the limited FPGA on-chip BRAM has enough capacity to load the weight parameters from the host memory due to the following reasons: (i) the regularity of the block circulant matrices introduces no additional storage such as weight indices after compression in ESE [18]; (ii) we use the heterogeneous weight quantization using ADMM considering hardware resource, further reducing the weight storage and exploiting the hardware resource while satisfying the accuracy requirement. The extra communication overhead caused by accessing FPGA off-chip DDR for common designs [18, 34] can be eliminated.

5.1 FPGA Resource-Aware Design Flow

The resource usage model including Look-up tables (LUTs), DSP blocks, and BRAM of an FPGA implementation can be estimated using analytical models. According to our design, there are two types of PEs: DSP-based PE for equal distance quantization and shift-based PE for mixed powers-of-two quantization. In the convolution operation, suppose the DSP resource for DSP-based and shift-based PE are ΔDSP_D and ΔDSP_S, respectively, and the LUT resouce for DSP-based and shift-based PE are ΔLUT_D and ΔLUT_S, respectively. The models of # DSP, # LUT, and # BRAM are shown as follows,

$$\#DSP = \Delta DSP_D \times \#CONV_D + \Delta DSP_S \times \#CONV_D \tag{6}$$

$$\#LUT = \Delta LUT_D \times \#CONV_L + \Delta LUT_S \times \#CONV_L \tag{7}$$

$$\#BRAM = max\{\frac{Model\ size}{BRAM\ size}, \frac{Onchip\ bandwidth}{BRAM\ bandwidth}\} \tag{8}$$

where $\#CONV_D$, $\#CONV_L$ are the number of CONV operations for DSP and LUT, respectively. Generally, in Xilinx Virtex-7 FPGA fabric, the BRAM size is 36kb and the BRAM bandwidth is 64b.

Indeed, replacing multiplications with bit shift operations significantly reduces the usage of DSP blocks, resulting in much less power assumption. However, the DSP resource will be wasted, causing utilization overhead on LUTs since LUT is the basic building block in implementing the logic function of bit shift operations. To fully exploit the limited FPGA resource for both LUTs and DSP blocks, we propose to adopt both the equal-distance quantization and the mixed powers-of-two-based quantization techniques for hardware implementation. More specifically, for each CONV layer, we select either equal-distance or mixed powers-of-two as the quantization method. Please note that the quantization method inside a CONV layer is identical.

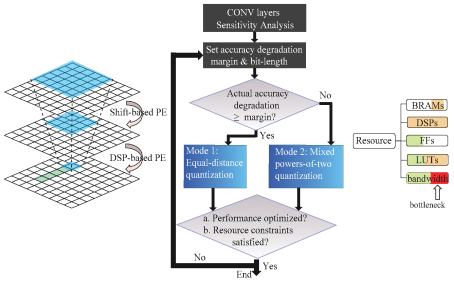

Figure 6: The mode selection rule of the resource-aware design.

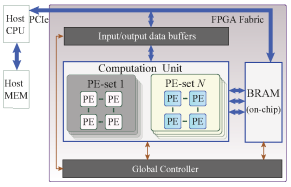

Figure 7: The overall hardware architecture on FPGA.

The selection rule is shown in Fig. 6. The design objectives are higher performance and energy efficiency satisfying the accuracy requirement. We first conduct the sensitivity analysis for each CONV layer regarding the two quantization methods (mode 1 and mode 2) and we set the initial margin for the overall degradation of the prediction accuracy and initial bit-length for weight representation. In order to reduce the accuracy degradation as much as possible, our priority choice is equal-distance quantization in those CONV layers which sensitivity are beyond the pre-set margin value since we can use DSPs for multiplication operations to enhance the accuracy. More specifically, we choose mode 2 if the actual accuracy degradation of the YOLO network is smaller than the margin value, otherwise we select mode 1. The margin value will further be refined until the performance is optimized and resource constraints (i.e., DSPs and bandwidth) are satisfied. Overall, the DSPs and LUTs usage is not the bottleneck in our design, which is different from traditional fixed-bit length weight quantization and our design is bound by bandwidth only.

5.2 Overall Hardware Architecture

Our proposed accelerator design on FPGA is composed of a computation unit, on-chip memories/BRAM, and datapath control logic. Our design does not access the FPGA off-chip memories. The FFT results of the pre-trained network model (CONV weight filters) is loaded to FPGA BRAM from the Host memory via the control of Host CPU and the PCI-express (PCIe) bus. The data buffers are used to cache input images, intermediate results and layer outputs from the previous stage slice by slice, to make preparation of the PE operation. The PEs inside of the computation unit are a collection of basic computing units that execute multiplications and additions (MACs), and other functions such as normalization, etc. The global controller orchestrates the computing flow and data flow on the FPGA fabric.

5.3 PE Design

From Eqn. (4), we observe that the FFT and IFFT operation are always executed in pairs. Therefore, we can combine and implement them as an FFT/IFFT kernel. An N-point IFFT calculation can be implemented using an N-point FFT in addition of a division operation (i.e., $\div N$) and two conjugations. There are N multipliers between FFT and IFFT, which is responsible for multiplying the intermediate results of FFT and weight values stored in BRAM. The PE is designed mainly to execute the most resource-consuming operation, i.e., matrix-vector multiplication, which will be implemented using the element-wise "FFT→MAC→IFFT" calculation according to Equation 4 (the key for the implementation with limited hardware resources). As shown in Fig. 8, the proposed PE architecture consists of a register bank, a controller, a weight decoder, a mode decoder, 2 multiplexers, and 2 FFT/IFFT kernels.

The register bank stores the FFT twiddle factors which will be loaded to the FFT operator. The weight decoder prepares the desired weight parameters format for further calculation. The mode decoder and MUX 1 work simultaneously to select the operating mode (i.e., mode 1 for equal-distance quantization and mode 2 for mixed powers-of-two quantization), under the control of the PE controller. Mux 2 is used to select the batch normalization (BN) operation depending on the layer structure. The MAC unit multiplies and accumulates the input feature maps and the pre-calculated FFT result of convolution kernel weights decoded from BRAM blocks. The two operating modes are marked in red dashed boxes, where mode 1 performs FFT/IFFT computations using multiplication-based FFT butterfly unit [8] along with a MAC unit, while mode 2 performs the FFT/IFFT computations using binary bit shifts and additions. Additionally, the MAC operation can be replaced by shift and addition in model 2.

5.4 Convolution Dataflow and Pipelining

The dataflow of input pixels from input buffers to computation unit to output bufferers is shown in Fig. 9. Taking advantage of the compressed but regular YOLO network structures, we apply inter-level and intra-level pipelining in the basic computation unit as shown in Fig. 9. In *intra-level pipelining*, there are three separate stages, i.e., load, compute, and store. This pipelining scheme generates higher level of parallelism and therefore leads to higher performance and throughput. In the *inter-level pipelining*, each pipeline stage corresponds to one level in the computation unit. There are several stages in the FFT operation depending on the input size, i.e., an

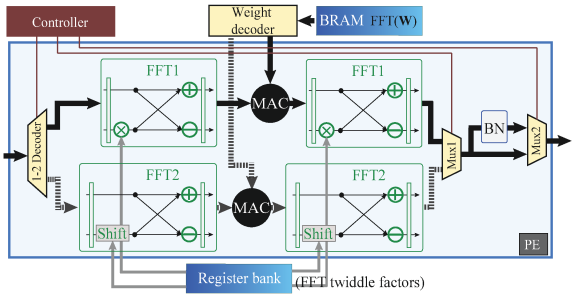

Figure 8: The PE (processing element) design.

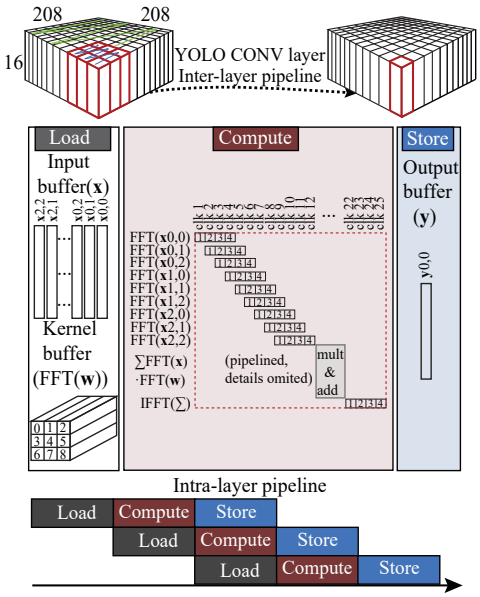

Figure 9: An illustration of the CONV operation data flow in the 2^{nd} CONV layer.

N-point FFT uses $N/2$ butterfly units for each stage and has a total of $r = \log_2^N$ stages.

We use an input size of $208 \times 208 \times 16$ and weight kernel size $3 \times 3 \times 16$ using 16-point FFT (4 stages) as shown in Fig. 9, to demonstrate the CONV dataflow in the 2^{th} CONV layer of the tiny YOLO network structure. Both inter-level and intra-level pipelining techniques are adopted. The input pixels are loaded to input buffers followed by a sequence corresponding to the spatial relationship with kernel window. The first input data needed for CONV operation is marked as red bars with the same input size of 16-point FFT. The PE accepts each input vector (red bar) each per clock cycle, computes the "FFT→MAC→IFFT" operation and the result is stored in the output buffer.

5.5 Design Optimization

In YOLO, the CONV operation, performed by PEs, is the most resource-intensive arithmetic. Therefore, from the perspective of computation, the hardware design optimization targets at PE size/number. Through reducing PE size/number, we can achieve less power and area consumption, leading to more available on-chip resource and more parallelism. From the communication perspective, the cost of moving data from one physical location to another on FPGAs, named communication cost, can dominate the computational energy and our design. Communication cost consists of accessing memory of weight parameters and intermediate results, and moving data bits over interconnect wires between PEs. Therefore, we can optimize the required computation and communication cost by reducing PE size/number including LUTs and DSPs, and memory access.

5.5.1 DSP Usage Optimization. Reducing the number of multiplications will be critical to the overall hardware design optimization. In the FFT operation, the multipliers of the Radix-2 FFT butterflies with twiddle factor 1 and -1 can be eliminated, and the multiplier

Algorithm 1: Pseudo-code for resource-aware exploration

Input: $\#CONV_D$, $\#CONV_L$, ΔDSP_D, ΔDSP_S, ΔLUT_D, ΔLUT_S, BRAM size and bandwidth, YOLO model Size, and onchip bandwidth.
Output: bit-length b, mode type of i_{th} layer $\mathbf{M}_i (i \in (0, \ldots, 8))$.
Analyze the sensitivity of all the CONV layers;
Set initial b & accuracy degradation margin ΔACC_m;
for $i \leftarrow 0$ *until* n_layers **do**
  $\mathbf{M}_i \leftarrow 2$;
  while *resource & performance not optimized* **do**
    Change b *or* ΔACC_m;
    if *actual accuracy loss* $\Delta ACC_{act} \geq \Delta ACC_m$ **then**
      $\mathbf{M}_i \leftarrow 1$;
    end
    Calculate resource usage #DSP, #LUT, and #BRAM using Equation (6, 7, 8);
  end
end
return b, $\mathbf{M}$.

of those butterflies with twiddle factor j and $-j$ can be replaced with conjugation operators. In the dot product stage, the two inputs of dot product are both from FFT operators, which are conjugate symmetric. And the dot product results of such conjugate symmetric inputs are also conjugate symmetric. Therefore, in $\left(\text{FFT}(\mathbf{x}_j) \circ \text{FFT}(\mathbf{w}_{ij})\right)$ with input size of FFT N, the last $N/2 - 1$ dot product outputs can be obtained using conjugation operations from their corresponding symmetric points. And the amount of $N/2 - 1$ multipliers can be eliminated.

The DSP48E1 block in modern Xilinx FPGAs generally consists of three sub-blocks: pre-adder, multiplier, and ALU. These hard blocks directly implement commonly used functions in silicon, therefore consuming much less power and area, and operating at a higher clock frequency than the same implementations in logic. For these hard blocks with constrained resource, resource sharing could be applied. Generally, non-overlapping MAC operations are scheduled using the combination of pre-adder, multiplier blocks or ALU, multiplier blocks based on the function itself and bit-length of operands.

In order to take full advantage of the limited DSP resource and achieve more design parallelism, we further optimize the proposed design using low-bit DSP sharing. More specifically, we can divide each sub-block into smaller slices, in which the internal carry propagation between slices is segregated to guarantee independence for all slices. In other word, we can group and feed several non-overlapping operands into one of the inputs of a DSP sub-block. For example, the ALU unit in DSP48E1 block can be divided into six 8-bit smaller slices with carry out signal for 8-bit computation.

5.5.2 Reducing Weight Memory Accesses. The input feature map $\mathbf{x}_i (i \in (1, \ldots, C))$ is real value [12] and all the weight parameters $\mathbf{W}$ are real-valued. According to [42, 6], the FFT or IFFT result is mirrored (have the property of complex conjugated symmetry) when its inputs are real-valued. Therefore, for an FFT/IFFT with N-point inputs, we only need to store $\frac{N}{2} + 1$ of the results instead of N results into the BRAM, thereby reducing communication energy.

5.6 Design Space Exploration

To prototype and explore the hardware architecture of the proposed REQ-YOLO framework, we use Xilinx SDx 2017.1 as the commercial synthesis backend to synthesize the C/C++ based YOLO LSTM design. We feed the well-trained inference models of YOLO into the automatic synthesis backend [28, 47, 48]. A bit-length of data quantization and mode selection for each CONV layer are generated to illustrate the computation flow as shown in Algorithm 1. The operators in each graph are scheduled to compose the intra-layer or inter-layer pipeline under the design constraints, to maximally achieve full throughput and performance. At last, a code generator receives the scheduling results and generates the final C/C++ based codes, which can be fed into the commercial HLS tool for FPGA implementation.

6 EVALUATION RESULTS

6.1 Training of Tiny YOLO

We adopt the state-of-the-art object detection algorithm-tiny YOLO based on yolov2 tiny [44] as the target DNN and evaluate it on both the PASCAL VOC07+12 dataset [12] and the DataDJI detection dataset captured by the DJI UAV [11]. We set S = 13, B =5. The DataDJI detection dataset has 12 labeled classes so C_{DJI} = 12, while the PASCAL VOC dataset has 20 labeled classes so C_{VOC} = 20. The anchor boxes sizes are pre-set by K-means clustering with K = 5. Our final prediction is a 13×13×37 tensor for DataDJI dataset and a 13×13×45 tensor for VOC dataset. After the last convolutional layer, 5 boxes for each grid cell will be obtained with their locations and scores. Then we first discard boxes that have detected a class with a score lower than the threshold, which is 0.6 in our experiment. After that, the Non-Maximum Suppression (NMS) algorithm will filter out remaining boxes that overlap with each other. The ideal output of YOLO is one bounding box for each object. Finally, for the predicted bounding box, we use *IOU* and mean average precision (*mAP*) as the metric to evaluate the object detection accuracy on DataDJI and VOC datasets, respectively.

For both datasets, we use a mini batch size of 16. The initial learning rate is set to 0.001, and divided by a factor of 10 every 20k iterations, to guarantee convergence. The ADAM [25] optimizer and standard data argumentation like random crops, color shifting, etc are used during training. The training results of the tiny YOLO using different block sizes are shown in Fig 10. Fig. 10 shows that the block circulant matrix-based training only causes a very small accuracy degradation (*IOU* or *mAP*) in general when the compression ratio is large. Among the six models, we select the YOLO-3 for the ADMM-based heterogeneous weight quantization and further hardware implementation, since it introduces small accuracy loss while maintaining the large compression ratio compared to baseline.

6.2 Accuracy after Weight Quantization

we select the YOLO-3 model with very small accuracy degradation and large weight reduction ratio for the REQ-YOLO framework and evaluate the selected model on both the PASCAL VOC dataset and the DataDJI detection dataset. For different weight (FFT results) representations from 32-bit to 6-bit, the introduced additional accuracy degradation (i.e., *IOU* for DataDJI and *mAP* for VOC) is generally very small, i.e., 0.73% for PASCAL, and 0.2% for DataDJI. Reducing the weight from 32-bit floating point to 8-bit fixed point brings

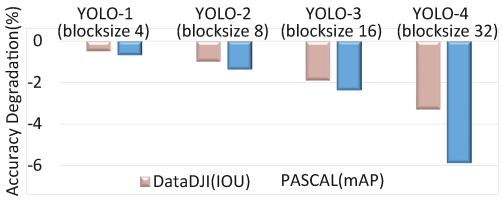

Figure 10: Test accuracy of the tiny YOLO network using different block sizes.

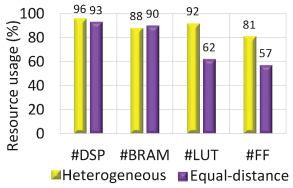

Figure 11: Resource usage comparison of two different quantization methods.

negligible additional degradation (0.07%) in *IOU* or *mAP* while the model size can further be compressed by 4×. In this way, through block circulant matrix training and ADMM-based heterogeneous quantization, we can accommodate the tiny YOLO structures to the on-chip BRAM of state-of-the-art FPGA while achieving real-time object detection, satisfying the accuracy requirement.

6.3 Performance and Energy Efficiency

6.3.1 FPGA-platform Comparison. We use the FPGA platform of Alpha Data's ADM-PCIE-7V3 for evaluating the proposed REQ-YOLO framework. The ADM-PCIE-7V3 board, comprising a Xilinx Virtex-7 (690t) FPGA and a 16GB DDR3 memory, is connected to the host machine through PCIE Gen3 × 8 I/O Interface. The host machine adopted in our experiments is a server configured with multiple Intel Core i7-4790 processors. The detailed comparison of on-chip resources of the FPGA platforms is presented in Fig. 11. We use Xilinx SDX 2017.4 as the commercial high-level synthesis backend to synthesize the high-level (C/C++) based RNN designs on the selected FPGAs. The REQ-YOLO framework of FPGA implementation is operating at 200MHz. For the tiny YOLO network, the performance of the first four layers is bound by the communication due to the large input/output feature map size. The last five layers of the YOLO network are otherwise constrained by the computation because of the increased channel size.

We conduct the comparison between the heterogeneous-based YOLO quantization method and the equal-distance-based quantization method on the selected ADM-7v3 platform. We report the resource usage (percentage) of two methods in Fig. 11. We can observe that the heterogeneous-based method better exploit the hardware resource than the equal-distance method, especially in LUT, therefore leading to higher performance and throughput. This finding also verifies our discussion in Section 5.1. The layer-wise computation, communication and latency analysis of both equal-distance quantization and heterogeneous quantization is shown in Table 1. In the communication-bound layers, the latency is the same for both methods. Overall, the heterogeneous-based method achieves 1.5× performance compared to the equal-distance method.

Table 1: Comparison of equal-distance-based quantization method and heterogeneous-based quantization method on the YOLO-3 model (block size 16).

Model	Layer	Comp. Size	Comm.		Bound Type	Equal-distance-based		Heterogeneous-based	
			In_size	Out_size		Latency (μ s)	Model Size	Latency (μ s)	Model Size
YOLO-3	*Conv0*	173,056	519,168	692,224	Comm.-bound	872.5	0.16kb	881.3	0.13kb
	Conv1	86,528	692,224	32,448	Comm.-bound	442.2	6.75kb	443.6	5.63kb
	Conv2	21,632	32,448	173,056	Comm.-bound	219.3	27.0kb	216.2	22.5kb
	Conv3	21,632	173,056	86,528	Comm.-bound	119.8	108.0kb	120.5	90.0kb
	Conv4	21,632	86,528	43,264	Comp.-bound	117.1	432.0kb	54.5	360.0kb
	Conv5	21,632	43,264	86,528	Comp.-bound	117.9	1.69Mb	69.4	1.41Mb
	Conv6	86,528	86,528	173,056	Comp.-bound	905.1	3.38Mb	430.4	2.81Mb
	Conv7	173,056	173,056	173,056	Comp.-bound	1,832.7	286.88kb	872.7	239.06kb
	Conv8	16,224	173,056	19,244	Comp.-bound	174.3	37.88kb	93.2	26.57kb
	Total	**621,920**	-	-	-	**4,801.0**	**5.93Mb**	**3,183.6**	**4.95Mb**

Table 2: Comparison among different tiny YOLO implementations.

Implementation	Titan X-YOLO [38]	Our GTX-YOLO	Our TX2-YOLO	Virtex-YOLO [33]	Zynq-YOLO [15]	**Our FPGA-YOLO0 (Equal-distance-based)**	**Our FPGA-YOLO1 (Heterogeneous-based)**
Device Type	Titan X	GTX 1070 GPU	TX2 embedded GPU	Xilinx Virtex-7 485t	Zynq 7020	ADM-7V3 FPGA	ADM-7V3 FPGA
Memory	12GB GDDR5	8GB GDDR5	8 GB LPDDR4	4.5 MB BRAM	0.6 MB BRAM	6.6 MB BRAM	6.6 MB BRAM
Clock Freq.	1.0 GHz	1.6 GHz	1.3 GHz	0.14 GHz	0.15 GHz (Peak)	0.2 GHz	0.2 GHz
Performance (FPS)	155	220.8	28.4	21	8	**208.2**	**314.2**
Power (W)	180	140	10.8	-	-	**23**	**21**
Energy Efficiency (FPS/W)	0.9	1.6	2.6	-	-	**9.1**	**15.0**

The results of performance and energy efficiency of our FPGA based YOLO implementations are presented in Table 2. Our FPGA-YOLO1 using heterogeneous quantization outperforms our FPGA-YOLO0 using equal-distance quantization in terms of both performance and energy efficiency, i.e., 1.5× in performance and 1.6× in energy efficiency, since the heterogeneous quantization fully exploits the hardware resource and design parallelism. Please note that since the DataDJI dataset is the latest released, we cannot find the related FPGA based implementations to compare with. For PASCAL VOC dataset, compared to other FPGA based works [15, 33], our FPGA-YOLO1 achieves at least 10× performance enhancement, while the FPGA fabric Virtex-7 690t in our platform is only slightly better than Virtex-7 485t used in [33] in resource capacity. We can not compare the energy efficiency among them since the power measurements are not provided in [15, 33].

6.3.2 Cross-platform Comparison. We implement the same YOLO network on two GPU platforms and compare with the tiny YOLO proposed in [38] using Titan X GPU. The first one is GeForce GTX 1070, which is a Nvidia GPU designed for PC. The second one is a Jetson TX2, which is the latest embedded GPU platform. The detailed specifications and comparisons among these platforms are shown in Table 2. We implement the trained model on both platforms and measure the performance using frame per second (FPS) and power consumption (W). Compared to Titan X-YOLO [38], our GTX-YOLO and our TX2-YOLO achieve 1.8× and 2.9× enhancement in energy efficiency.

Compared to GPU-based YOLO implementation (Our GTX-YOLO), our two FPGA YOLO implementations has the similar or better speed while dissipating around 6× less power, and the efficiency (performance per power) of our FPGA-YOLO0 and our FPGA-YOLO1 are 5.7× and 9.4× better, respectively. It indicates that our proposed REQ-YOLO framework is very suitable for FPGAs, since usually GPUs often perform faster than FPGAs as discussed in [3]. Compared to the GPU-based YOLO implementation with the best energy efficiency (our TX2-YOLO), our two FPGA YOLO implementations achieve 3.5× and 5.8× improvement in energy efficiency. While our FPGA YOLO implementations are at least 7.3× faster while only dissipating at most 2.1× more power.

Overall, our proposed REQ-YOLO framework is effective on both GPUs and FPGAs. It is highly promising to deploy our proposed REQ-YOLO framework on FPGA to gain much higher energy efficiency for autonomous systems on object sections than on GPUs. More importantly, the proposed framework achieves much higher FPS over the real-time requirement.

7 CONCLUSION

In this work, we propose REQ-YOLO, a resource-aware, systematic weight quantization framework for object detection, considering both algorithm and hardware resource aspects in object detection. We adopt the block-circulant matrix method and we incorporate ADMM with FFT/IFFT and develop a heterogeneous weight quantization method including both equal-distance and heterogeneous quantization methods considering hardware resource. We implement the quantized models on the state-of-the-art FPGA taking advantage of the potential to store the whole compressed DNN models on-chip. To achieve real-time, highly-efficient implementations on FPGA, we develop an efficient PE structure supporting both equal-distance and mixed powers-of-two quantization methods, CONV dataflow and pipelining techniques, design optimization techniques focus on reducing memory access and PE size/numbers, and a template-based automatic synthesis framework to optimally exploit hardware resource. Experimental results show that our proposed framework can significantly compress the YOLO model while introducing very small accuracy degradation. Our framework is very suitable for FPGA and our FPGA implementations outperform the state-of-the-art designs.

ACKNOWLEDGMENTS

This work is supported by Beijing Natural Science Foundation (No. L172004), Municipal Science and Technology Program under Grant

Z181100008918015, and National Science Foundation under grants CNS #1704662 and CNS #1739748. We thank all the anonymous reviewers for their feedback.

REFERENCES

[1] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-layer CNN accelerators. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 1–12.

[2] Yakoub Bazi and Farid Melgani. 2018. Convolutional SVM Networks for Object Detection in UAV Imagery. *IEEE Transactions on Geoscience and Remote Sensing*, 56, 6, 3107–3118.

[3] Brahim Betkaoui, David B Thomas, and Wayne Luk. 2010. Comparing performance and energy efficiency of FPGAs and GPUs for high productivity computing. In *IEEE FPT'10*.

[4] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, Jonathan Eckstein, et al. 2011. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trendső in Machine learning*, 3, 1, 1–122.

[5] Zhaowei Cai, Quanfu Fan, Rogerio S Feris, and Nuno Vasconcelos. 2016. A unified multi-scale deep convolutional neural network for fast object detection. In *European conference on computer vision*. Springer, 354–370.

[6] Yun-Nan Chang and Keshab K Parhi. 2003. An efficient pipelined fft architecture. *Ieee transactions on circuits and systems ii: analog and digital signal processing*, 50, 6, 322–325.

[7] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. 2017. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52, 1, 127–138.

[8] James W Cooley and John W Tukey. 1965. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19, 90, 297–301.

[9] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. Binaryconnect: training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, 3123–3131.

[10] Caiwen Ding et al. 2017. CirCNN: Accelerating and Compressing Deep Neural Networks using Block-circulant Weight Matrices. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, 395–408.

[11] DJI. 2018. http://www.cse.cuhk.edu.hk/byu/2018-DAC-HDC. (2018).

[12] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. 2010. The pascal visual object classes (voc) challenge. *International journal of computer vision*, 88, 2, 303–338.

[13] Ross Girshick. 2015. Fast R-CNN. In *Proceedings of the IEEE international conference on computer vision*, 1440–1448.

[14] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. 2014. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the ieee conference on computer vision and pattern recognition*, 580–587.

[15] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Song Yao, Song Han, Yu Wang, and Huazhong Yang. 2016. From model to FPGA: Software-hardware Co-design for Efficient Neural Network Acceleration. In *Hot chips 28 symposium (hcs), 2016 ieee*. IEEE, 1–27.

[16] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 243–254.

[17] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: compressing deep neural networks with pruning, trained quantization and huffman coding. *Arxiv preprint arxiv:1510.00149*.

[18] Song Han et al. 2017. Ese: efficient speech recognition engine with sparse lstm on fpga. In *Fpga*. ACM, 75–84.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the ieee conference on computer vision and pattern recognition*, 770–778.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2014. Spatial pyramid pooling in deep convolutional networks for visual recognition. In *European conference on computer vision*. Springer, 346–361.

[21] Donald R High and Noah Ryan Kapner. 2018. Apparatus and method for providing unmanned delivery vehicles with expressions. US Patent App. 15/638,960. (Jan. 2018).

[22] Sergey Ioffe. 2017. Batch renormalization: towards reducing minibatch dependence in batch-normalized models. In *Advances in neural information processing systems*, 1945–1953.

[23] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: accelerating deep network training by reducing internal covariate shift. *Arxiv preprint arxiv:1502.03167*.

[24] Rong Jin. 2017. Deep learning at alibaba. In *Proceedings of the 26th international joint conference on artificial intelligence*. AAAI Press, 11–16.

[25] Diederik P Kingma and Jimmy Ba. 2014. Adam: a method for stochastic optimization. *Arxiv preprint arxiv:1412.6980*.

[26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*.

[27] Yann LeCun. 2015. Lenet-5, convolutional neural networks. *Url: http://yann. lecun. com/exdb/lenet*.

[28] Yun Liang et al. 2012. High-level Synthesis: Productivity, Performance, and Software Constraints. *JECE'12*.

[29] Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. 2016. Fixed point quantization of deep convolutional networks. In *International conference on machine learning*, 2849–2858.

[30] Zhouhan Lin, Matthieu Courbariaux, Roland Memisevic, and Yoshua Bengio. 2015. Neural networks with few multiplications. *Arxiv preprint arxiv:1510.03009*.

[31] Liqiang Lu and Yun Liang. 2018. SpWA: An Efficient Sparse Winograd Convolutional Neural Networks Accelerator on FPGAs. In *DAC'18*.

[32] Liqiang Lu, Yun Liang, Qingcheng Xiao, and Shengen Yan. 2017. Evaluating Fast Algorithms for Convolutional Neural Networks on FPGAs. In *FCCM'17*.

[33] Jing Ma, Li Chen, and Zhiyong Gao. 2017. Hardware implementation and optimization of tiny-yolo network. In *International forum on digital tv and wireless multimedia communications*. Springer, 224–234.

[34] Hiroki Nakahara, Haruyoshi Yonekawa, Tomoya Fujii, and Shimpei Sato. 2018. A lightweight yolov2: a binarized cnn with a parallel support vector regression for an fpga. In *Proceedings of the 2018 acm/sigda international symposium on field-programmable gate arrays*. ACM, 31–40.

[35] Victor Pan. 2012. *Structured matrices and polynomials: unified superfast algorithms*. Springer Science & Business Media.

[36] Jiantao Qiu et al. 2016. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 acm/sigda international symposium on field-programmable gate arrays*. ACM, 26–35.

[37] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. Xnor-net: imagenet classification using binary convolutional neural networks. In *European conference on computer vision*. Springer, 525–542.

[38] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You only look once: unified, real-time object detection. In *Proceedings of the ieee conference on computer vision and pattern recognition*, 779–788.

[39] Joseph Redmon and Ali Farhadi. 2017. Yolo9000: better, faster, stronger. *Arxiv*.

[40] Joseph Redmon and Ali Farhadi. 2018. Yolov3: an incremental improvement. *Arxiv preprint arxiv:1804.02767*.

[41] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2015. Faster r-cnn: towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, 91–99.

[42] Sayed Ahmad Salehi, Rasoul Amirfattahi, and Keshab K Parhi. 2013. Pipelined architectures for real-valued fft and hermitian-symmetric ifft with real datapaths. *Ieee transactions on circuits and systems ii: express briefs*, 60, 8, 507–511.

[43] Julius Orion Smith. 2007. *Mathematics of the discrete fourier transform (dft): with audio applications*. Julius Smith.

[44] Trieu. 2016. https://github.com/AlexeyAB/darknet. (2016).

[45] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. 2017. Finn: a framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 acm/sigda international symposium on field-programmable gate arrays*. ACM, 65–74.

[46] Shuo Wang, Zhe Li, Caiwen Ding, Bo Yuan, Qinru Qiu, Yanzhi Wang, and Yun Liang. 2018. C-LSTM: Enabling Efficient LSTM Using Structured Compression Techniques on FPGAs. In *Fpga'18*.

[47] Shuo Wang and Yun Liang. 2017. A Comprehensive Framework for Synthesizing Stencil Algorithms on FPGAs using OpenCL Model. In *DAC'17*.

[48] Shuo Wang, Yun Liang, and Wei Zhang. 2017. FlexCL: An Analytical Performance Model for OpenCL Workloads on Flexible FPGAs. In *DAC'17*.

[49] Xuechao Wei, Yun Liang, Xiuhong Li, Cody Hao Yu, Peng Zhang, and Jason Cong. 2018. TGPA: Tile-grained Pipeline Architecture for Low Latency CNN Inference. In *ICCAD'18*.

[50] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. 2017. Automated systolic array architecture synthesis for high throughput cnn inference on fpgas. In *Proceedings of the 54th annual design automation conference 2017*. ACM, 29.

[51] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning structured sparsity in deep neural networks. In *Advances in neural information processing systems*, 2074–2082.

[52] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Chen. 2016. Quantized convolutional neural networks for mobile devices. In *Computer vision and pattern recognition, 2016. cvpr 2016. ieee conference on*.

[53] Qingcheng Xiao, Yun Liang, Liqiang Lu, Shengen Yan, and Yu-Wing Tai. 2017. Exploring Heterogeneous Algorithms for Accelerating Deep Convolutional Neural Networks on FPGAs. In *DAC'17*.

[54] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 acm/sigda international symposium on field-programmable gate arrays*. ACM, 161–170.

[55] Liang Zhao, Siyu Liao, Yanzhi Wang, Zhe Li, Jian Tang, and Bo Yuan. 2017. Theoretical properties for neural networks with weight matrices of low displacement rank. In *International conference on machine learning*, 4082–4090.