YOLObile: Real-Time Object Detection on Mobile Devices via Compression-Compilation Co-Design

Yuxuan Cai[1*], Hongjia Li[1*], Geng Yuan[1*], Wei Niu[2], Yanyu Li[1],
Xulong Tang[3], Bin Ren[2], Yanzhi Wang[1]
[1]Northeastern University
[2]William & Mary
[3]University of Pittsburgh
[1]{cai.yuxu, li.hongjia, yuan.geng, li.yanyu, yanz.wang}@northeastern.edu,
[2]wniu@email.wm.edu, [2]bren@cs.wm.edu, [3]tax6@pitt.edu

Abstract

The rapid development and wide utilization of object detection techniques have aroused attention on both accuracy and speed of object detectors. However, the current state-of-the-art object detection works are either accuracy-oriented using a large model but leading to high latency or speed-oriented using a lightweight model but sacrificing accuracy. In this work, we propose YOLObile framework, a real-time object detection on mobile devices via compression-compilation co-design. A novel block-punched pruning scheme is proposed for any kernel size. To improve computational efficiency on mobile devices, a GPU-CPU collaborative scheme is adopted along with advanced compiler-assisted optimizations. Experimental results indicate that our pruning scheme achieves 14× compression rate of YOLOv4 with 49.0 mAP. Under our YOLObile framework, we achieve 17 FPS inference speed using GPU on Samsung Galaxy S20. By incorporating our proposed GPU-CPU collaborative scheme, the inference speed is increased to 19.1 FPS, and outperforms the original YOLOv4 by 5× speedup. Source code is at: https://github.com/nightsnack/YOLObile.

1 Introduction

Object detection, one of the major tasks in the computer vision field, has been drawing extensive research from both academia and industry thanks to the breakthrough of deep neural network (DNN). Object detection is widely adopted in numerous computer vision tasks, including image annotation, event detection, object tracking, segmentation, and activity recognition, with a wide range of applications, such as autonomous driving, UAV obstacle avoidance, robot vision, human-computer interaction, and augmented reality. Considering these application scenarios, it is equivalently important to maintain high accuracy and low latency simultaneously when deploy such applications on resource-limited platforms, especially mobiles and embedded devices.

In the past decades, promising object detection approaches are proposed, which are mainly categorized into two-stage detectors (Girshick et al. 2014; Girshick 2015; Ren et al. 2015; He et al. 2017) and one-stage detectors (Redmon et al. 2016; Redmon and Farhadi 2017, 2018; Bochkovskiy, Wang, and Liao 2020; Liu et al. 2016; Lin et al. 2017). Compared with two-stage detectors, one-stage detectors aim to provide an equitable trade-off between accuracy and speed, and will be mainly discussed in this work. Despite large efforts devoted, representative works such as You Only Look Once (YOLO) (Redmon et al. 2016; Redmon and Farhadi 2017, 2018; Bochkovskiy, Wang, and Liao 2020), Single Shot Detector (SSD) (Liu et al. 2016), still require extensive computation to achieve high mean average precision (mAP), result in the main limitation for real-time deployment on mobile devices. Apart from large-scale approaches mentioned above, lightweight object detection architectures targeted for mobile devices are investigated (Sandler et al. 2018a; Huang, Pedoeem, and Chen 2018; Li et al. 2018). However, the accomplished efficiency leads to non-negligible accuracy drop.

To address this issue in object detection detectors, model compression techniques have been drawing attention, especially weight pruning methods, which have been proved as one of the most effective approaches to reduce extensive computation and memory intensity without sacrificing accuracy (Wen et al. 2016; Guo, Yao, and Chen 2016; Min et al. 2018; He et al. 2018, 2019). By reducing the vast redundancy in the number of weights, models with structural sparsity achieve higher memory and power efficiency and low latency during inference. Generally, unstructured pruning and structured pruning are the two main trendy schemes of weight pruning. Unstructured pruning eliminates weights in an irregular manner, which causes the essential drawback to obstruct hardware accelerations (Han et al. 2015; Guo, Yao, and Chen 2016; Liu et al. 2018). Structured pruning is observed with notable accuracy degradation due to the coarse-grained nature in pruning whole filters/channels (Min et al. 2018; Zhuang et al. 2018; Zhu, Zhou, and Li 2018; Ma et al. 2020b; Zhao et al. 2019; Liu et al. 2020b). To overcome these shortcomings, pattern-based pruning is proposed incorporating fine-grained unstructured pruning in a hardware aware fashion (Ma et al. 2020a; Niu et al. 2020). However, it is only applicable to convolutional (CONV) layers with 3×3 kernels, significantly limiting its deployment in object

[*] These Authors contributed equally.

detection tasks.

The goal of this paper is to achieve real-time object detection by exploiting the full advantages of pruning for inference on mobile platforms. We propose YOLObile framework, a real-time object detection on mobile devices via compression-compilation co-design. State-of-the-art detectors YOLOv4 (Bochkovskiy, Wang, and Liao 2020) is adopted as our detection architecture. YOLObile consists of several novel techniques including a new block based pruning for arbitrary kernel size, compiler optimizations, and a GPU-CPU collaborative acceleration strategy. To be more specific, we propose a novel pruning scheme—-block-punched pruning, which is flexible and applicable to CONV layers with *any* kernel size as well as fully-connected (FC) layers. The whole DNN weights from a certain layer are divided into a number of equal-sized blocks and the weights within a block are pruned to a same shape. To improve the computational efficiency of DNNs on mobile devices, the proposed YOLObile also adopts a GPU-CPU collaborative computation scheme. As a result, YOLObile achieves high hardware parallelism using our proposed compiler optimizations, including compact storage scheme, block reordering, and highly parallel auto-tuning model employment. Experimental results indicate that YOLObile delivers 14× compression rate (in weights) of YOLOv4 with 49.0 mAP. It achieves 19.1 frames per second (FPS) inference speed on an off-the-shelf Samsung Galaxy S20, and is 5× faster than the original YOLOv4.

2 Background

2.1 Preliminaries on Object Detection DNNs

The DNN-based object detectors can be categorized into two mainstreams: (i) two-stage detectors and (ii) one-stage detectors.

Two-stage detectors divide the detection to two stages: extract region of interest (RoI) and then do the classification and bounding box regression tasks based on the RoI. A most representative series of two-stage detectors is R-CNN (Girshick et al. 2014) with its extended generations Fast R-CNN (Girshick 2015) and Faster R-CNN (Ren et al. 2015). R-CNN is the first region-based CNN object detector and it achieves higher object detection performance compared with previous HOG-like features-based systems (Liu et al. 2020a). Through ensuing development, Faster R-CNN has improved both precision and detection efficiency. Despite highest accuracy rates achieved, the major drawback of such two-stage detectors is high computation and still relatively slower inference speed due to the two-stage detection procedure.

One-stage detectors eliminate the RoI extraction stage and directly classify and regress the candidate anchor boxes. YOLO (Redmon et al. 2016) adopts a unified architecture that extracts feature maps from input images, then it regards the whole feature maps as candidate regions to predict bounding boxes and categories. YOLOv2 (Redmon and Farhadi 2017), YOLOv3 (Redmon and Farhadi 2018) and YOLOv4 (Bochkovskiy, Wang, and Liao 2020) are proposed with improved speed and precision. SSD (Liu et al. 2016) is another representative one-stage detector, establishing a fully convolution network to predict a fixed numbers of bounding boxes and scores. One-stage detectors demonstrate an optimized trade-off between accuracy and speed only on high performance desktop GPUs.

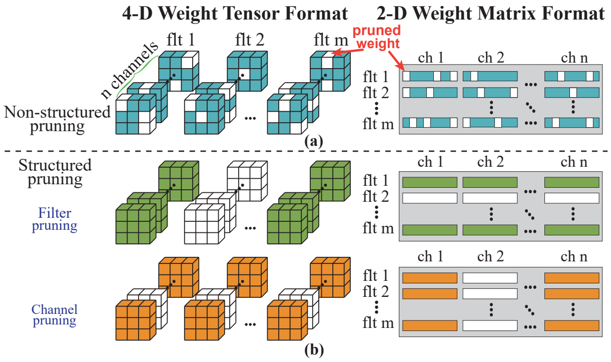

Figure 1: Illustration of (a) unstructured pruning and (b) coarse-grained structured pruning.

Therefore, lightweight object detectors are proposed for mobile devices where both model size and speed are limited. SSDLite (Sandler et al. 2018b) is a mobile friendly variant of regular SSD utilizing a mobile architecture, MobileNet, which is based on an inverted residual structure and uses lightweight depthwise convolutions to filter features. Based on YOLOv2, YOLO-LITE (Huang, Pedoeem, and Chen 2018) provides a smaller, faster, and more efficient model increasing the accessibility of real-time object detection to a variety of devices. Tiny-DSOD (Li et al. 2018) is based on a deeply supervised object detection (DSOD) framework and dedicates to resource-restricted usages by adopting depthwise dense block (DDB) based backbone and depthwise feature-pyramid-network (D-FPN) based front-end. However, the accuracy of these works is sacrificed significantly. On COCO dataset under AP@[0.5:0.95] metric, the accuracy of YOLO-LITE decreases to 12.16 (Huang, Pedoeem, and Chen 2018), while SSDLite (Sandler et al. 2018b) achieves only 22.1 and Tiny-DSOD obtains 23.2 (Li et al. 2018).

2.2 DNN Model Pruning

We now discuss the three most trendy pruning schemes: including fine-grained unstructured pruning, coarse-grained structured pruning, and pattern-based pruning.

Unstructured pruning allows the weights at arbitrary locations in the weight matrix to be pruned, which ensures a higher flexibility to search for optimized pruning structure (Guo, Yao, and Chen 2016; Frankle and Carbin 2018; Dai, Yin, and Jha 2019), as shown in Figure 1 (a). Thus, it usually achieves high compression rate with minor accuracy loss. However, unstructured pruning leads to an irregular sparsity in the weight matrix, which requires additional indices to locate the non-zero weights during the computation. This makes the hardware parallelism provided by the underlying system (e.g., GPUs in mobile platforms) underutilized. Consequently, the unstructured pruning is not appli-

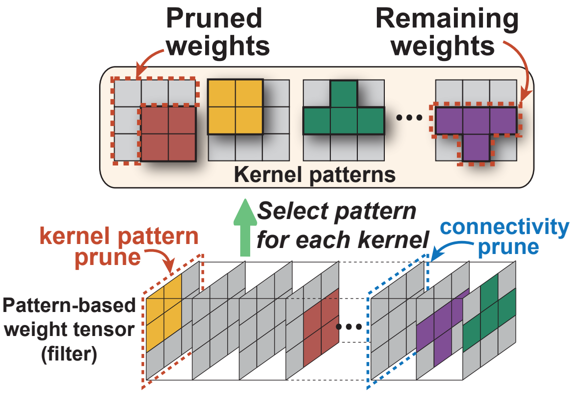

Figure 2: Illustration of pattern-based pruning.

cable for DNN inference acceleration, and even a decrease in speed can be observed (Wen et al. 2016).

Structured pruning prunes the entire channel(s)/filter(s) of DNN weights (Wen et al. 2016; He, Zhang, and Sun 2017; He et al. 2019; Yu et al. 2018). As Figure 1 (b) shows, the filter pruning removes whole row(s) of the weight matrix, where the channel pruning prunes the consecutive columns of corresponding channel(s) in the weight matrix. Structured pruning maintains the regular shape of the weight matrix with reduced dimension. Therefore, it is hardware friendly and can leverage the hardware parallelism to facilitate acceleration. However, structured pruning suffers from considerable accuracy loss due to its coarse-grained pruning feature.

Pattern-based pruning is considered as a fine-grained structured pruning scheme. It simultaneously preserves the accuracy and the hardware performance due to its proper degree of structural flexibility and structural regularity (Niu et al. 2020; Ma et al. 2020a). Pattern-based pruning consists of two parts, which are the kernel pattern pruning and the connectivity pruning. The kernel pattern pruning prunes a fixed number of weights in each convolution kernel, as shown in Figure 2. It first analyzes the locations of remaining weights and forms a specific kernel pattern, then prunes the weights accordingly. Different kernels can apply different types of patterns but the total number of patterns is restricted to a fixed-size set. Connectivity pruning, as a supplementary of the kernel pattern pruning, is adopted to achieve higher overall compression rate. Connectivity pruning prunes the entire convolution kernels, which can be considered as removing the connections between certain input and output channels. However, kernel patterns are specially designed for 3×3 kernels and are not applicable to other kernel sizes. This drawback significantly restricts the use of pattern-based pruning in many scenarios.

2.3 Compiler-assisted DNN Acceleration on Mobile

With the growth of mobile vision application, there is a growing need to break through the current performance limitation of mobile platforms. Both industry and academia have been putting efforts on mobile DNN execution frameworks (e.g., (Lane et al. 2016; Lane, Georgiev, and Qendro 2015; Xu et al. 2018; Huynh, Lee, and Balan 2017; Yao et al. 2017; Han et al. 2016)). Among these works, TensorFlow-Lite (TFLite) (Google 2020), Alibaba Mobile Neural Network (MNN) (Alibaba 2020), and TVM (Chen et al. 2018) are three representative end-to-end DNN execution frameworks with high execution efficiency. Advanced performance optimization techniques are employed, including varied computation graph optimizations, tensor optimizations, half-float support; particularly, TVM adopts a more advanced parameters auto-tuning. However, none of those frameworks provide support for sparse (pruned) DNN models on mobile platforms[1]. This significantly limits the performance of the DNN inference on mobile devices.

To overcome this drawback, a set of compiler-based optimizations are proposed to support sparse DNN models, significantly accelerating the end-to-end DNN inference on mobile devices. However, these optimizations are designed for fine-grained pattern-based pruning such as PatDNN (Niu et al. 2020) and PCONV (Ma et al. 2020a), in which the specific 3×3 convolution kernels in CONV layers are the main acceleration part. In addition, commonly used layers in object detection such as FC layers and 1×1 CONV layers are not supported.

3 Motivation

As mentioned above, the state-of-the-art object detection works are either accuracy-oriented using a large model size (Ren et al. 2015; Liu et al. 2016; Bochkovskiy, Wang, and Liao 2020) or speed-oriented using a lightweight model but sacrificing accuracy (Sandler et al. 2018a; Huang, Pedoeem, and Chen 2018; Li et al. 2018). Therefore, all of them are hard to simultaneously satisfy the accuracy and latency demands of practical applications on mobile devices. As a result,

> We need a solution that can achieve both high accuracy and low latency on mobile devices.

While pattern-based pruning seems to be a desirable option since it strikes a balance between execution efficiency and accuracy. However, it is only applicable to the 3×3 CONV layers and hinders its effectiveness in object detection tasks. Figure 3 illustrates the comparison between 3 × 3 convolution layers and non 3 × 3 layers. We select 4 representative object detection approaches and compare the percentage of their weights and computations. For example, in YOLOv4, one of the representative state-of-the-art networks for object detection, a considerable number of weights and amount of computations (17% and 19%, respectively) are contributed by non-3×3 CONV layers.

Compiler-assisted DNN inference acceleration is another attractive option for low-latency DNN inference on mobile devices. It has been proved that, with the assistance of compiler optimizations, the low latency DNN inference can be achieved for image classification tasks (Niu et al. 2020; Ma et al. 2020a). However, such acceleration is still not sufficient enough to satisfy the low latency required by object detection tasks, since it has massive amount of weights and

[1] TVM considers sparsity recently for desktop processors.

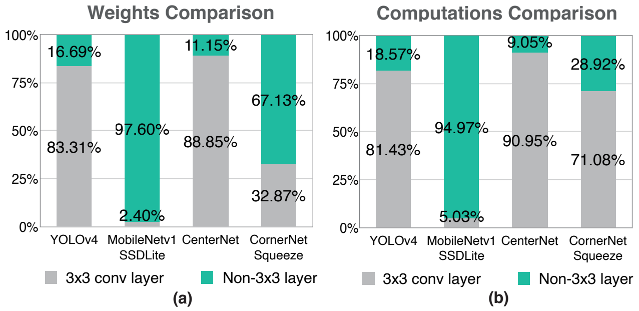

Figure 3: Comparisons of (a) weights ratio and (b) computations ratio for 3×3 convolution layers and non-3×3 layers for different object detection approach.

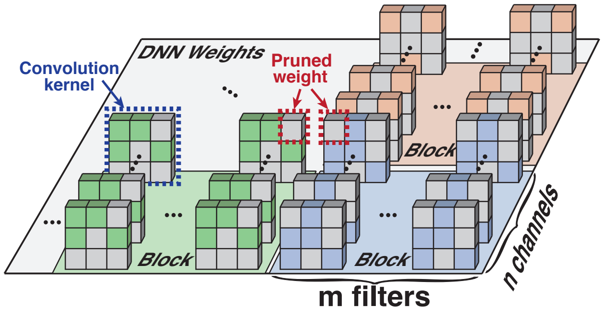

Figure 4: Illustration of block-punched pruning.

requires more complex computations (Bochkovskiy, Wang, and Liao 2020; Liu et al. 2016). To this end, we raise two key design objectives:

- **Objective 1:** We need a pruning scheme that can (i) simultaneously achieve high accuracy and leverage the underlying hardware parallelism, and (ii) be ubiquitously applied on different types of layers.
- **Objective 2:** We need a more efficient computation method for further accelerating the DNN inference speed of object detection tasks.

4 Framework Design

4.1 Block-Punched Pruning

To achieve the first objective in Section 3, we propose a novel pruning scheme–block-punched pruning, which preserves high accuracy while achieving high hardware parallelism. In addition to the 3×3 CONV layer, it can also be mapped to other types of DNN layers, such as 1×1 CONV layer and FC layer. Moreover, it is particularly suitable for high-efficient DNN inference on resource-limited mobile devices. As shown in Figure 4, the whole DNN weights from a certain layer are divided to a number of equal-sized blocks, where each block contains the weights from n consecutive channels of m consecutive filters. In each block, we prune a group of weights at the same location of all filters while also pruning the weights at the same location of all channels. In other words, the weights to be pruned will punch through the same location of all filters and channels within a block. Note that the number of pruned weights in each block is flexible and can be different across different blocks.

From the accuracy perspective, inspired by the pattern-based pruning (Niu et al. 2020), we adopt a fine-grained structured pruning strategy in block-punched pruning to increase structural the flexibility and mitigate accuracy loss.

From the hardware performance perspective, compared to the coarse-grained structured pruning, our block-punched pruning scheme is able to achieve high hardware parallelism by leveraging the appropriate block size and the help of compiler-level code generation. The reason is that typically the number of weights in a DNN layer is very large. Even when we divide the weights into blocks, the computation required by each block is still sufficient to saturate hardware computing resources and achieve high degree of parallelism, especially on the resource-limited mobile devices. Moreover, our pruning scheme can better leverage the hardware parallelism from both memory and computation perspectives. First, during convolution computation, all filters in each layer share the same input. Since the same locations are pruned among all the filters within each block, these filters will skip reading the same input data, thus mitigating the memory pressure among the threads processing these filters. Second, the restriction of pruning identical locations across channels within a block ensures that all of these channels share the same computation pattern (indices), thus eliminating the computation divergence among the threads processing the channels within each block.

In our block-punched pruning, block size affects both the accuracy and the hardware acceleration. On the one hand, a smaller block size provides higher structural flexibility due to its finer granularity, which typically achieves higher accuracy, but at the cost of reduced speed. On the other hand, larger block size can better leverage the hardware parallelism to achieve higher acceleration, but it may cause more severe accuracy loss.

To determine an appropriate block size, we first determine the number of channels contained in each block by considering the computation resource of the device. For example, we use the same number of channels for each block as the length of the vector registers in the mobile CPU/GPU on a smartphone to achieve high parallelism. If the number of channels contained in each block is less than the length of the vector registers, both the vector registers and vector computing units will be underutilized. On the contrary, increasing the number of channels will not gain extra on the performance but cause more severe accuracy drop. Thus, the number of filters contained in each block should be determined accordingly, considering the trade-off between accuracy and hardware acceleration.

The hardware acceleration can be inferred by the inference speed, which can be obtained without the need of retraining the DNN model and is easier to derive compared with model accuracy. Thus, a reasonable minimum required inference speed is set as the design target that needs to be satisfied. As long as the block size satisfies the inference speed target, we choose to keep the smallest number of filters in each block to mitigate the accuracy loss. More detailed results will be elaborated in Section 5.3.

4.2 Reweighted Regularization Pruning Algorithm

In the previous weight pruning algorithms, methods such as group Lasso regularization (Wen et al. 2016; He, Zhang, and Sun 2017; Liu et al. 2017) or Alternating Direction Methods of Multipliers (ADMM) (Zhang et al. 2018; Ren et al. 2019; Li et al. 2019) are mainly adopted. However, it leads to either potential accuracy loss or requirement of manual compression rate tuning. Therefore, we adopt the reweighted group Lasso (Candes, Wakin, and Boyd 2008) method. The basic idea is to systematically and dynamically reweight the penalties. To be more specific, the reweighted method reduces the penalties on weights with larger magnitudes, which are likely to be more critical weights, and increases the penalties on weights with smaller magnitudes.

Let $\boldsymbol{W}_i \in \mathbb{R}^{M\times N\times K_h\times K_w}$ denote the 4-D weight tensor of the i-th CONV layer of CNN, where M is the number of filters; N is the number of input channels; K_w and K_h are the width and height kernels of i-th layer. The general reweighted pruning problem is formulated as

$$\underset{\boldsymbol{W},\boldsymbol{b}}{\text{minimize}} \quad f\big(\boldsymbol{W};\boldsymbol{b}\big) + \lambda\sum_{i=1}^{N} R(\boldsymbol{\alpha}_i^{(t)}, \boldsymbol{W}_i), \tag{1}$$

where $f\big(\boldsymbol{W};\boldsymbol{b}\big)$ represents loss function of DNN. $R(\cdot)$ is the regularization term used to generate model sparsity and the hyperparameter λ controls the trade-off between accuracy and sparsity. $\boldsymbol{\alpha}_i^{(t)}$ denotes the collection of penalty values applied on the weights $\boldsymbol{W}_i$ for layer i.

Under our block-punched pruning, each $\boldsymbol{W}_i$ is divided into K blocks with the same size $g_i m \times g_i n$, namely, $\boldsymbol{W}_i = [\boldsymbol{W}_{i1}, \boldsymbol{W}_{i2}, ..., \boldsymbol{W}_{iK}]$, where $\boldsymbol{W}_{ij} \in \mathbb{R}^{g_i m\times g_i n}$. Therefore, the regularization term is

$$R(\boldsymbol{\alpha}_i^{(t)}, \boldsymbol{W}_i) = \sum_{j=1}^{K}\sum_{h=1}^{g_m^i}\sum_{w=1}^{g_n^i} \big\|\alpha_{ijn}^{(t)} \circ [\boldsymbol{W}_{ij}]_{h,w}\big\|_F^2, \tag{2}$$

where $\alpha_{ijn}^{(t)}$ is updated by $\alpha_{ijn}^{(t)} = \frac{1}{\|[\boldsymbol{W}_{ij}]_{h,w}^t\|_F^2+\epsilon}$.

The pruning process starts with a pre-trained DNN model. By conducting another training process using the reweighted regularization pruning algorithm, the pruned model with our block-punched constraints can be obtained.

4.3 Mobile Acceleration with a Mobile GPU-CPU Collaborative Scheme

To achieve the second objective in Section 3, we propose a GPU-CPU collaborative computation scheme to improve the computational efficiency of DNNs on mobile devices. It can be observed that the multi-branch architecture, as shown in Figure 5 (a), are widely used in many state-of-the-art networks such as YOLOv4. Mobile devices has mobile GPU and mobile CPU, currently the DNN inference acceleration frameworks such as TFLite and MNN can only support DNN inference to be executed on either the mobile GPU or the CPU sequentially, which leads to a potential waste of the computation resources. The CPU is underutilized for most of the time when the GPU is computing. Moreover, many branches have no dependencies on each other, and potentially could be computed on mobile GPU and mobile CPU concurrently to achieve higher efficiency and speed.

In our framework, we incorporate our GPU-CPU collaborative computation scheme to optimize two types of branch structures in DNNs, which are 1) the branch structure with CONV layers and 2) the branch structure with non-CONV operations. The examples of the two types of branch structures are shown in Figure 5 (a) and (b). We do the offline device selection based on the speed before deployment.

As we know, the GPU is suitable for high-parallelism computation, such as the convolutional computations, and it significantly outperforms the CPU in terms of speed. Thus, ***for the branch structure with CONV layers***, such as the Cross Stage Partial (CSP) block in YOLOv4 as shown in Figure 5 (a), the GPU is selected for computing the most time-consuming branch, and the problem left is to determine whether the other branches use CPU to compute concurrently or still use GPU to compute sequentially.

In Figure 5 (a), we name the GPU computing time in branch 1 and branch 2 as t_{g1}, t_{g2}, CPU computing time as t_{c1}, t_{c2}, and data copying time as τ. We execute the most time-consuming branch 1 in GPU and make a decision for branch 2. When using CPU for parallel computing, we also need to add the data copying time τ. The desired GPU-CPU parallel computing time T_{par} depends on the maximum time cost of the branch 1 and branch 2:

$$T_{par} = max\{t_{g1}\,,\, t_{c2} + \tau\}$$

The GPU-only serial computing time T_{ser} is the summation of computing time $t_{g1} + t_{g2}$ of two branches:

$$T_{ser} = t_{g1} + t_{g2}$$

Based on the minimum of GPU-CPU parallel computing time T_{par} and GPU-only computing time T_{ser}, we can select the optimal executing device for branch 2. Note that the determination of execution devices for each branch structure in YOLOv4 is independent to other branch structures. Thus, the execution devices for all branch structures in the network can be solved by greedy algorithm (Cormen et al. 2009).

On the other hand, limited by the power and area, mobile GPUs usually have lower performance. For the less computational intensive operations, such as point-wise add operation and point-wise multiplication operation, mobile CPU performs similar or even faster speed compared with mobile GPU. Therefore, ***for the branch structures with non-CONV operations***, either of CPU or GPU can be used for each branch depending on total computation time.

Take the three final output YOLO head structures in YOLOv4 as an example, as shown in Figure 5 (b). After transposing and reshaping the output from the last CONV layer in each branch, we still need several non-CONV operations to get the final output. We measure the total GPU and CPU execution times for the non-CONV operations in each branch and denote them as t_{g0}, t_{g1}, t_{g2} and t_{c0}, t_{c1}, t_{c2} respectively. The T_{total} denotes the total computing time for all three branches.

Now we have eight possible combinations of device selections for the three branches. For example, if first two

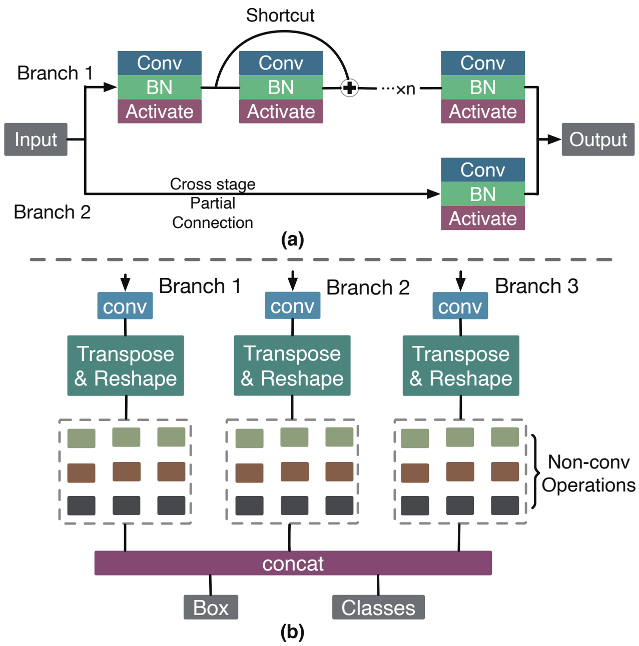

Figure 5: An illustration of the (a) cross-stage partial block and (b) non-convolutional operations in YOLO Head.

branches use CPU and the third branch uses GPU, the total computing time will be

$$T_{total} = max\{t_{c0} + t_{c1} \, , \, t_{g2}\}$$

Note that the final output has to be moved to CPU sooner or later, so we do not count the data copying time into the total computation time. As a result, we select the combination that has the minimum total computation time as our desired computation scheme. Putting all together, our proposed GPU-CPU collaborative scheme can effectively increase the hardware utilization and improve the DNN inference speed.

4.4 Compiler-assisted Acceleration

Inspired by PatDNN (Niu et al. 2020), YOLObile relies on several advanced compiler-assisted optimizations that are enabled by our newly designed block-punched pruning to further improve the inference performance. We summarize them here briefly due to the space constraints. First, YOLObile stores the model weights compactly by leveraging the pruning information (the block and punched pattern) that can further compress the index arrays comparing to the well-known Compressed Sparse Row format. Second, YOLObile reorders blocks to improve memory and computation regularity, and to eliminate unnecessary memory access. Moreover, YOLObile employs a highly parallel auto-tuning model to find the best execution configuration parameters. YOLObile generates both CPU and GPU codes for each layer, and calls the right one according to our GPU-CPU collaborative scheme during the actual inference process.

#Weights	#Weights Comp. Rate	#FLOPs	mAP	AP@[.5:.95]	FPS
64.36M	1×	35.8G	57.3	38.2	3.5
16.11M	3.99×	10.48G	55.1	36.5	7.3
8.04M	8.09×	6.33G	51.4	33.3	11.5
6.37M	10.1×	5.48G	50.9	32.8	13
4.59M	14.02×	3.95G	49	31.9	17

Table 1: Accuracy and speed under different compression rates.

5 Evaluation

In this section we evaluate our proposed YOLObile framework on mobile devices in terms of accuracy and inference speed, compared with other state-of-the-art frameworks. Additionally, ablation study on different pruning schemes and configurations are provided.

Experimental Setup Our models are trained on a server with eight NVIDIA RTX 2080Ti GPUs. The training methods are implemented using PyTorch API. We evaluate our framework on an off-the-shelf Samsung Galaxy S20 smartphone, which has a Qualcomm Snapdragon 865 Octa-core CPU and a Qualcomm Adreno 650 GPU. Each test runs on 50 different input frames (images), with the average speed results reported. Our YOLObile is derived based on YOLOv4, with 320×320 input size, and train on MS COCO dataset (Lin et al. 2014). We denote mAP as the Average Precision under IoU 0.5 threshold and AP@[.5:.95] as the Average Precision under IoU from 0.5 to 0.95. Note that our compiler achieves much higher speed for object detection approaches compared with existing compiler-assisted frameworks, such as TFLite. More comparisons are presented in supplementary materials.

5.1 Evaluation of block-punched pruning

We first evaluate the accuracy and compression rate of our proposed block-punched pruning in YOLObile framework. As mentioned above, block size affects both accuracy and hardware acceleration performance. We adopt 8×4 as our block size, i.e. 4 consecutive channels of 8 consecutive filters. The details of the impact of different block sizes are discussed in ablation study 5.3. The original YOLOv4 model contains 64.36M weights and requires 35.8G floating-point operations (FLOPs). As shown in Table 1, by applying our block-punched pruning, we achieve the compression rate up to 14× (in weights) with 49 mAP. The weight number decreases to 4.59M and FLOPs is reduced to 3.59G. With 92.87% weights and 88.97% FLOPs reduced, our model still maintains a decent accuracy, with only 8.3 mAP loss.

5.2 Evaluation of YOLObile framework

To validate the effectiveness of our framework, we compare our YOLObile with several representative works. To make fair comparisons, all the results (including the object detection approaches from the reference works) are evaluated under our compiler optimizations. As shown in Table 2, under a similar number of computations and even having a smaller

Approach	Input Size	backbone	#Weights	#FLOPs	mAP	AP@[.5:.95]	FPS
CenterNet-DLA (Duan et al.)	512	DLA34	16.9M	52.58G	57.1	39.2	1.9
CornerNet-Squeeze (Law et al.)	511	-	31.77M	150.15G	-	34.4	0.3
SSD (Liu et al.)	300	VGG16	26.29M	62.8G	43.1	25.1	4.2
MobileNetv1-SSDLite (Sandler et al.)	300	MobileNetv1	4.31M	2.30G	-	22.2	49
MobileNetv2-SSDLite (Sandler et al.)	300	MobileNetv2	3.38M	1.36G	-	22.1	41
Tiny-DSOD (Li et al.)	300	-	1.15M	1.12G	40.4	23.2	-
YOLOv4 (Bochkovskiy, Wang, and Liao)	320	CSPDarknet53	64.36M	35.5G	57.3	38.2	3.5
YOLO-Lite (Huang, Pedoeem, and Chen)	224	-	0.6M	1.0G	-	12.26	36
YOLOv3-tiny (Redmon and Farhadi)	320	Tiny Darknet	8.85M	3.3G	29	14	14
YOLOv4-tiny (Bochkovskiy, Wang, and Liao)	320	Tiny Darknet	6.06M	4.11G	40.2	-	11
YOLObile (GPU only)	320	CSPDarknet53	4.59M	3.95G	**49**	**31.6**	**17**
YOLObile (GPU&CPU)	320	CSPDarknet53	4.59M	3.95G	**49**	**31.6**	**19.1**

Table 2: Accuracy (mAP) and speed (FPS) comparison with other object detection approaches.

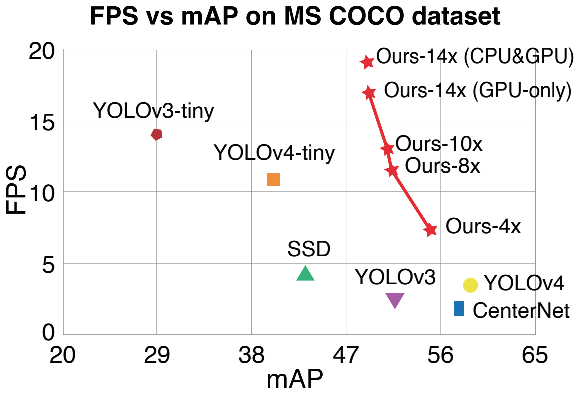

Figure 6: The accuracy (mAP) and speed (FPS) comparison of YOLObile under different compression rate and different approaches.

Pruning Scheme	#Weights	#Weights Comp. Rate	mAP	FPS
Not Prune	64.36M	1×	57.3	3.5
Unstructured	8.04M	8.09×	53.9	6.4
Structured	8.04M	8.09×	38.6	13
Ours	8.04M	8.09×	51.4	11.5

Table 3: Comparison of different pruning schemes.

model size, YOLObile consistently outperforms YOLOv3-tiny and YOLOv4-tiny in terms of mAP and FPS. This indicates our proposed block-punched pruning is a more desired method to achieve a smaller model size while maintaining the mAP compared to training a small model from scratch.

YOLObile achieves even higher mAP than the full-size one-stage detector SSD but lower mAP than YOLOv4 and CenterNet. However, the inference speed of YOLObile is much faster than SSD, YOLOv4 and CenterNet (4.5×, 5.5× and 10× respectively). Comparing with the lightweight detectors such as YOLO-Lite and MobileNetv2-SSDLite, YOLObile has lower FPS but much higher mAP. Figure 6 demonstrates the mAP and FPS of YOLObile under different compression rates and the results are compared with representative reference works. Our YOLObile lies in top right of the figure, and outperforms YOLOv3, SSD, YOLOv3-tiny and YOLOv4-tiny in both accuracy and speed. Unlike the lightweight approaches, which simply trade the mAP for FPS, YOLObile provides a Pareto-Optimal trade-off solution that maintains both the mAP and FPS.

We also evaluate the performance of our GPU-CPU collaborative computation scheme. As shown in Table 2, comparing to the GPU-only execution, our GPU-CPU collaborative computation scheme effectively accelerates the inference speed and improves FPS.

5.3 Ablation Study

Ablation study on pruning scheme. In this section, we conduct experiments on YOLOv4 under different pruning schemes. Table 3 shows the comparison of different pruning scheme results under 8× compression rate. Unstructured pruning scheme achieves the highest mAP because of its flexibility. However, the inference speed in FPS is only 6.4 due to underutilized hardware parallelism. Structured pruning (filter pruning) shows high inference speed, but with severe accuracy drop. Compared with structured pruning and unstructured pruning, our block-punched pruning scheme achieves both high accuracy and fast inference speed.

Ablation Study on pattern-based pruning. Despite the promising performance that pattern-based pruning can achieve, it has the restriction of kernel size to be 3, while in non-3×3 layers it cannot be applied. Unfortunately, in object detection approaches such as YOLOv4, the ratio of 3×3 CONV layers is 83.31% in total weights, which limits the highest compression rate of pattern-based pruning to 5.99×. Therefore, we compare pattern-based pruning and our block-punched pruning scheme under compression rate of 2×, 3×, 4×, 5×, respectively. Additionally, we demonstrate the result under our block-punched pruning scheme with 8×, 10× and 14 × compression rate. In Figure 7, we plot the mAP curve and FPS bar under different compression rate. We can see when the compression rate is below 3× pattern-based pruning has higher accuracy as it is more flexible. When the compression rate increases, for pattern-based pruning, we have to prune more weights in each 3×3 CONV

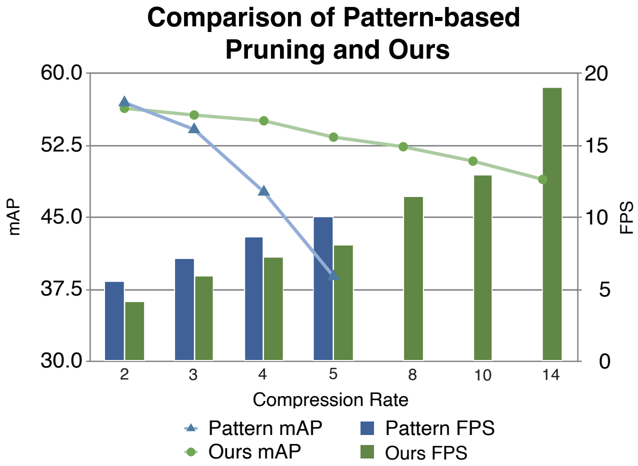

Figure 7: mAP and FPS comparison of pattern-based pruning and ours.

layer, because non-3×3 layers can not be pruned. The extremely high layer-wise prune ratio results in a sharp drop down of the curve. Pattern-based pruning has higher inference speed compared to our block-punched pruning. However, the compression rate ceiling limits the inference speed. Ours scheme can reach higher speed when compression rate increases, and ours do not suffer from sharply accuracy drop down.

Ablation study on block size. We conduct experiments on four different block sizes using our block-punched pruning scheme to check the impact of block size on results. To achieve high hardware parallelism, the number of channels in each block is fixed to 4, which is the same as the length of GPU and CPU's vector registers. The accuracy and speeds are evaluated under different numbers of filters in each block. As shown in Figure 8, larger block size can better leverage the hardware parallelism compared with smaller block size and achieves higher inference speed. However, it leads to accuracy loss due to its coarse pruning granularity. Smaller block size can achieve higher accuracy but sacrifice the inference speed. According to the results, we consider 8×4 (4 consecutive channels of 8 consecutive filters) as a desired block size on mobile devices, which strikes a good balance between both the accuracy and the speed.

Ablation study on layer-wise compression rate between different kernel size. YOLOv4 contains only 3×3 kernel size and 1×1 kernel size in CONV layers. We believe these 2 types of CONV layers have different levels of sensitivity in pruning process, therefore we conduct 2 groups of experiments. Under the same number of FLOPs, we evenly prune all the layers in one group. And in another group, the compression rate of 3×3 CONV layers is 1.15× higher than in 1×1 CONV layers. As shown in Table 4, the evenly pruned model exhibits lower accuracy and lower inference speed than the unevenly pruned model. Since 3×3 CONV layers contributes 81.4% of the total FLOPs, it can be concluded that compression rates in these layers illustrate a higher impact on the overall performance.

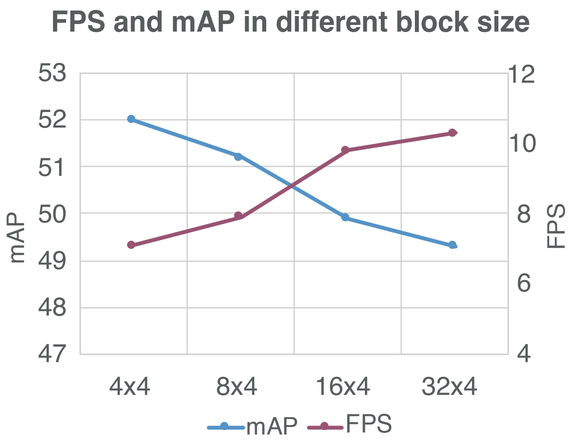

Figure 8: Accuracy (mAP) and speed (FPS) of different block size pruning results.

Pruning method	#Weights	#FLOPs	mAP	FPS
Evenly Prune	10.38M	6.2G	50.5	8
Unevenly Prune	8.04M	6.2G	51.4	11.5

Table 4: Evenly Prune vs Unevenly Prune.

6 Conclusion

In this work, we propose YOLObile, a real-time object detection framework on mobile devices via compression-compilation co-design. A novel pruning scheme—-block-punched pruning is also proposed, designed for CONV layers with *any* kernel size as well as fully-connected (FC) layers. To improve the computational efficiency of DNNs on mobile devices, the proposed YOLObile also features a GPU-CPU collaborative computation scheme in addition to our proposed compiler optimizations. The evaluation demonstrates that our YOLObile framework exhibits high accuracy while achieving high hardware parallelism.

References

Alibaba. 2020. https://github.com/alibaba/MNN.

Bochkovskiy, A.; Wang, C.-Y.; and Liao, H.-Y. M. 2020. YOLOv4: Optimal Speed and Accuracy of Object Detection. *arXiv preprint arXiv:2004.10934* .

Candes, E. J.; Wakin, M. B.; and Boyd, S. P. 2008. Enhancing sparsity by reweighted ℓ_1 minimization. *Journal of Fourier analysis and applications* 14(5-6): 877–905.

Chen, T.; Moreau, T.; Jiang, Z.; Zheng, L.; Yan, E.; Shen, H.; Cowan, M.; Wang, L.; Hu, Y.; Ceze, L.; Guestrin, C.; and Krishnamurthy, A. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; and Stein, C. 2009. *Introduction to algorithms*. MIT press.

Dai, X.; Yin, H.; and Jha, N. K. 2019. NeST: A neural network synthesis tool based on a grow-and-prune paradigm. *IEEE Transactions on Computers* 68(10): 1487–1497.

Duan, K.; Bai, S.; Xie, L.; Qi, H.; Huang, Q.; and Tian, Q. 2019. Centernet: Keypoint triplets for object detection. In *Proceedings of the IEEE International Conference on Computer Vision*, 6569–6578.

Frankle, J.; and Carbin, M. 2018. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *The International Conference on Learning Representations (ICLR)*.

Girshick, R. 2015. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, 1440–1448.

Girshick, R.; Donahue, J.; Darrell, T.; and Malik, J. 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.

Google. 2020. https://www.tensorflow.org/lite.

Guo, Y.; Yao, A.; and Chen, Y. 2016. Dynamic network surgery for efficient dnns. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Han, S.; Pool, J.; Tran, J.; and Dally, W. 2015. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems (NeurIPS)*.

Han, S.; Shen, H.; Philipose, M.; Agarwal, S.; Wolman, A.; and Krishnamurthy, A. 2016. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 123–136. ACM.

He, K.; Gkioxari, G.; Dollár, P.; and Girshick, R. 2017. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, 2961–2969.

He, Y.; Lin, J.; Liu, Z.; Wang, H.; Li, L.-J.; and Han, S. 2018. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*.

He, Y.; Liu, P.; Wang, Z.; Hu, Z.; and Yang, Y. 2019. Filter pruning via geometric median for deep convolutional neural networks acceleration. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

He, Y.; Zhang, X.; and Sun, J. 2017. Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*.

Huang, R.; Pedoeem, J.; and Chen, C. 2018. YOLO-LITE: a real-time object detection algorithm optimized for non-GPU computers. In *2018 IEEE International Conference on Big Data (Big Data)*, 2503–2510. IEEE.

Huynh, L. N.; Lee, Y.; and Balan, R. K. 2017. Deepmon: Mobile gpu-based deep learning framework for continuous vision applications. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 82–95. ACM.

Lane, N. D.; Bhattacharya, S.; Georgiev, P.; Forlivesi, C.; Jiao, L.; Qendro, L.; and Kawsar, F. 2016. Deepx: A software accelerator for low-power deep learning inference on mobile devices. In *Proceedings of the 15th International Conference on Information Processing in Sensor Networks*, 23. IEEE Press.

Lane, N. D.; Georgiev, P.; and Qendro, L. 2015. DeepEar: robust smartphone audio sensing in unconstrained acoustic environments using deep learning. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, 283–294. ACM.

Law, H.; Teng, Y.; Russakovsky, O.; and Deng, J. 2019. Cornernet-lite: Efficient keypoint based object detection. *arXiv preprint arXiv:1904.08900* .

Li, T.; Wu, B.; Yang, Y.; Fan, Y.; Zhang, Y.; and Liu, W. 2019. Compressing convolutional neural networks via factorized convolutional filters. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Li, Y.; Li, J.; Lin, W.; and Li, J. 2018. Tiny-DSOD: Lightweight object detection for resource-restricted usages. *arXiv preprint arXiv:1807.11013* .

Lin, T.-Y.; Goyal, P.; Girshick, R.; He, K.; and Dollár, P. 2017. Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*, 2980–2988.

Lin, T.-Y.; Maire, M.; Belongie, S.; Hays, J.; Perona, P.; Ramanan, D.; Dollár, P.; and Zitnick, C. L. 2014. Microsoft coco: Common objects in context. In *European conference on computer vision*, 740–755. Springer.

Liu, L.; Ouyang, W.; Wang, X.; Fieguth, P.; Chen, J.; Liu, X.; and Pietikäinen, M. 2020a. Deep learning for generic object detection: A survey. *International journal of computer vision* 128(2): 261–318.

Liu, N.; Ma, X.; Xu, Z.; Wang, Y.; Tang, J.; and Ye, J. 2020b. AutoCompress: An Automatic DNN Structured Pruning Framework for Ultra-High Compression Rates. In *AAAI*.

Liu, W.; Anguelov, D.; Erhan, D.; Szegedy, C.; Reed, S.; Fu, C.-Y.; and Berg, A. C. 2016. SSD: Single Shot MultiBox Detector. In *ECCV*.

Liu, Z.; Li, J.; Shen, Z.; Huang, G.; Yan, S.; and Zhang, C. 2017. Learning efficient convolutional networks through network slimming. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*.

Liu, Z.; Sun, M.; Zhou, T.; Huang, G.; and Darrell, T. 2018. Rethinking the Value of Network Pruning. In *International Conference on Learning Representations*.

Ma, X.; Guo, F.-M.; Niu, W.; Lin, X.; Tang, J.; Ma, K.; Ren, B.; and Wang, Y. 2020a. Pconv: The missing but desirable sparsity in dnn weight pruning for real-time execution on mobile devices. In *Thirty-Fourth AAAI conference on artificial intelligence (AAAI)*.

Ma, X.; Yuan, G.; Lin, S.; Ding, C.; Yu, F.; Liu, T.; Wen, W.; Chen, X.; and Wang, Y. 2020b. Tiny but Accurate: A Pruned, Quantized and Optimized Memristor Crossbar Framework for Ultra Efficient DNN Implementation. In *ASP-DAC*.

Min, C.; Wang, A.; Chen, Y.; Xu, W.; and Chen, X. 2018. 2pfpce: Two-phase filter pruning based on conditional entropy. *arXiv preprint arXiv:1809.02220* .

Niu, W.; Ma, X.; Lin, S.; Wang, S.; Qian, X.; Lin, X.; Wang, Y.; and Ren, B. 2020. Patdnn: Achieving real-time DNN execution on mobile devices with pattern-based weight pruning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

Redmon, J.; Divvala, S.; Girshick, R.; and Farhadi, A. 2016. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 779–788.

Redmon, J.; and Farhadi, A. 2017. YOLO9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 7263–7271.

Redmon, J.; and Farhadi, A. 2018. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767* .

Ren, A.; Zhang, T.; Ye, S.; Li, J.; Xu, W.; Qian, X.; Lin, X.; and Wang, Y. 2019. Admm-nn: An algorithm-hardware codesign framework of dnns using alternating direction methods of multipliers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 925–938.

Ren, S.; He, K.; Girshick, R.; and Sun, J. 2015. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, 91–99.

Sandler, M.; Howard, A.; Zhu, M.; Zhmoginov, A.; and Chen, L.-C. 2018a. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Sandler, M.; Howard, A.; Zhu, M.; Zhmoginov, A.; and Chen, L.-C. 2018b. MobileNetV2: Inverted Residuals and Linear Bottlenecks. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* doi:10.1109/cvpr.2018.00474. URL http://dx.doi.org/10.1109/CVPR.2018.00474.

Wen, W.; Wu, C.; Wang, Y.; Chen, Y.; and Li, H. 2016. Learning structured sparsity in deep neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Xu, M.; Zhu, M.; Liu, Y.; Lin, F. X.; and Liu, X. 2018. DeepCache: Principled Cache for Mobile Deep Vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, 129–144. ACM.

Yao, S.; Hu, S.; Zhao, Y.; Zhang, A.; and Abdelzaher, T. 2017. Deepsense: A unified deep learning framework for time-series mobile sensing data processing. In *Proceedings of the 26th International Conference on World Wide Web*, 351–360.

Yu, R.; Li, A.; Chen, C.-F.; Lai, J.-H.; Morariu, V. I.; Han, X.; Gao, M.; Lin, C.-Y.; and Davis, L. S. 2018. Nisp: Pruning networks using neuron importance score propagation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Zhang, T.; Ye, S.; Zhang, K.; Tang, J.; Wen, W.; Fardad, M.; and Wang, Y. 2018. A systematic dnn weight pruning framework using alternating direction method of multipliers. In *Proceedings of the European Conference on Computer Vision (ECCV)*.

Zhao, C.; Ni, B.; Zhang, J.; Zhao, Q.; Zhang, W.; and Tian, Q. 2019. Variational convolutional neural network pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Zhu, X.; Zhou, W.; and Li, H. 2018. Improving Deep Neural Network Sparsity through Decorrelation Regularization. In *Ijcai*, 3264–3270.

Zhuang, Z.; Tan, M.; Zhuang, B.; Liu, J.; Guo, Y.; Wu, Q.; Huang, J.; and Zhu, J. 2018. Discrimination-aware channel pruning for deep neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*.