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

Yuxuan Cai *
Northeastern University
cai.yuxu@northeastern.edu

Geng Yuan *
Northeastern University
yuan.geng@northeastern.edu

Hongjia Li *
Northeastern University
li.hongjia@northeastern.edu

Wei Niu
College of William and Mary
wniu@email.wm.edu

Yanyu Li
Northeastern University
li.yanyu@northeastern.edu

Xulong Tang
University of Pittsburgh
tax6@pitt.edu

Bin Ren
College of William and Mary
bren@cs.wm.edu

Yanzhi Wang
Northeastern University
yanz.wang@northeastern.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.

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 impor-

*These Authors contributed equally.

Machine Learning for Autonomous Driving Workshop at the 34th Conference on Neural Information Processing Systems (NeurIPS 2020), Vancouver, Canada.

tant 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 [10, 9, 37, 14] and one-stage detectors [33, 34, 35, 2, 26, 22]. 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) [33, 2], Single Shot Detector (SSD) [26], 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 [38, 18, 21]. 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 [40, 12, 31, 16, 15]. 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 [13, 12, 28]. Structured pruning is observed with notable accuracy degradation due to the coarse-grained nature in pruning whole filters/channels [31, 45, 44, 30, 43, 25]. To overcome these shortcomings, pattern-based pruning is proposed incorporating fine-grained unstructured pruning in a hardware aware fashion [29, 32]. However, it is only applicable to convolutional (CONV) layers with 3×3 kernels, significantly limiting its deployment in object 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. Our contributions are summarized as follows:

- We propose block-punched pruning, a novel pruning scheme combines the advantage of high accuracy and the capability of achieving high hardware-parallelism.
- We propose a GPU-CPU collaborative computation scheme to compute DNN branch structures in a more efficient fashion, which further improves the inference speed.
- Under our compiler-assisted optimization, with comparable accuracy to state-of-the-art object detectors, our proposed framework achieves real-time object detection on mobile devices.

Experimental results indicate that YOLObile delivers $14\times$ 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 $5\times$ 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 [10] with its extended generations Fast R-CNN [9] and Faster R-CNN [37]. 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 [24]. 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 [33] 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 [34], YOLOv3 [35] and YOLOv4 [2] are proposed with improved speed and precision.

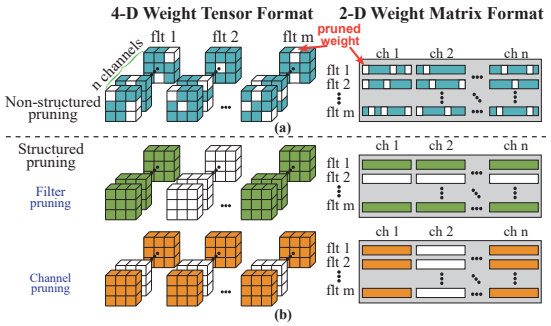

Figure 1: Illustration of (a) unstructured pruning and (b) coarse-grained structured pruning.

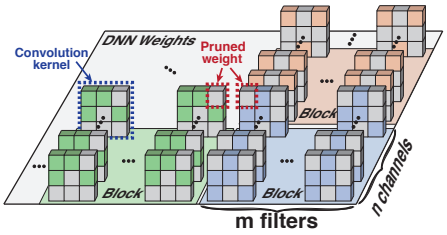

Figure 2: Illustration of block-punched pruning.

One-stage detectors demonstrate an optimized trade-off between accuracy and speed only on high performance desktop GPUs.

Therefore, lightweight object detectors are proposed for mobile devices where both model size and speed are limited. SSDLite [39] is a mobile friendly variant of regular SSD utilizing the backbone of MobileNet. Based on YOLOv2, YOLO-LITE [18] provides a smaller, faster, and more efficient model increasing the accessibility of real-time object detection to a variety of devices. However, the accuracy of these works is sacrificed significantly.

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 [12, 8, 6], as shown in Figure 1 (a). Thus, it usually achieves high compression rate with minor accuracy loss. Though it has the privilege of achieving high compression rate with minor accuracy loss, the irregular sparsity in the weight matrix 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 applicable for DNN inference acceleration, and even a decrease in speed can be observed [40].

Structured pruning prunes the entire channel(s)/filter(s) of DNN weights [40, 17, 15, 41]. 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 [32, 29], which simultaneously preserves the accuracy and the hardware performance. Pattern-based pruning prunes weights by enforcing the locations of the remaining weights in a 3×3 convolutional kernel to form a specific kernel pattern. 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. TensorFlow-Lite (TFLite) [11], Alibaba Mobile Neural Network (MNN) [1], and TVM [4] are three representative end-to-end DNN execution frameworks that support mobile platforms and have high execution efficiency. However, none of those frameworks provide support for sparse (pruned) DNN models on mobile platforms[2]. This significantly limits the

[2]TVM considers sparsity recently for desktop processors.

performance of the DNN inference on mobile devices. Thus, a set of compiler-based optimizations are proposed to support sparse DNN models in PatDNN [32] and PCONV [29]. However, they only support the pattern-based pruning on 3×3 convolutional (CONV) layers, where the commonly used layers such as FC layers and 1×1 CONV layers are still not supported. 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 requires more complex computations [2, 26].

3 Framework Design

3.1 Block-Punched Pruning

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 2, 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 [32], 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 4.3.

3.2 Reweighted Regularization Pruning Algorithm

In the previous weight pruning algorithms, methods such as group Lasso regularization [40, 17, 27] or Alternating Direction Methods of Multipliers (ADMM) [42, 36, 20] 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 [3] 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

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

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.

3.3 Mobile Acceleration with a Mobile GPU-CPU Collaborative Scheme

To improve the computational efficiency of DNNs on mobile devices and meet the low latency requirements of complex object detection task, we propose a GPU-CPU collaborative computation scheme.

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. It can be observed that the multi-branch architecture, as shown in Figure 3 (a), are widely used in many state-of-the-art networks such as YOLOv4. 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. 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 3 (a), the GPU is selected for computing the most time-consuming branch, and the problem

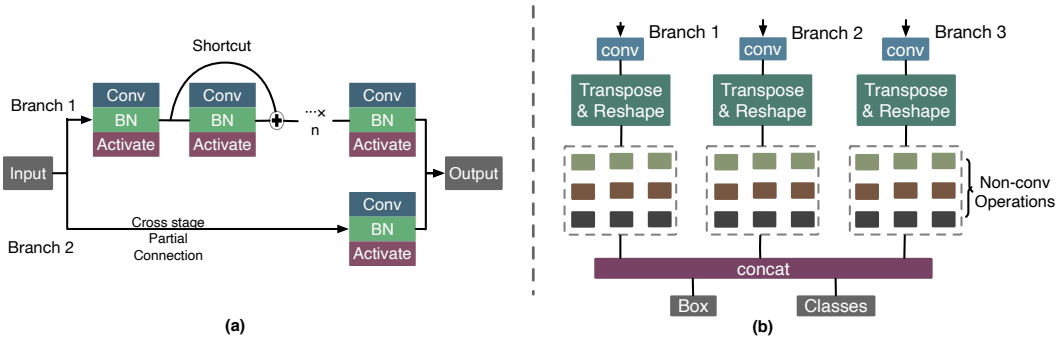

Figure 3: An illustration of the (a) cross-stage partial block and (b) non-convolutional operations in YOLO Head.

left is to determine whether the other branches use CPU to compute concurrently or still use GPU to compute sequentially.

In Figure 3 (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 [5].

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 3 (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 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 hardware utilization and improve the inference speed.

3.4 Compiler-assisted Acceleration

Inspired by PatDNN [32], 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

#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.

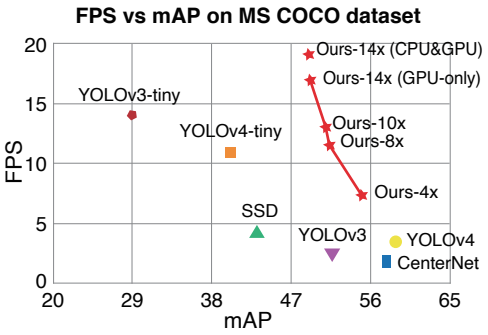

Figure 4: The accuracy (mAP) and speed (FPS) comparison of YOLObile under different compression rate and different approaches.

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.

4 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. 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 [23]. 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.

4.1 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 4.3. 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.

4.2 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 model size, YOLObile consistently outperforms YOLOv3-tiny and YOLOv4-tiny in terms of mAP and FPS. This indicates our proposed

Approach	Input Size	backbone	#Weights	#FLOPs	mAP	AP@[.5:.95]	FPS
CenterNet-DLA ([7])	512	DLA34	16.9M	52.58G	57.1	39.2	1.9
CornerNet-Squeeze ([19])	511	-	31.77M	150.15G	-	34.4	0.3
SSD ([26])	300	VGG16	26.29M	62.8G	43.1	25.1	4.2
MobileNetv1-SSDLite ([38])	300	MobileNetv1	4.31M	2.30G	-	22.2	49
MobileNetv2-SSDLite ([38])	300	MobileNetv2	3.38M	1.36G	-	22.1	41
Tiny-DSOD ([21])	300	-	1.15M	1.12G	40.4	23.2	-
YOLOv4 ([2])	320	CSPDarknet53	64.36M	35.5G	57.3	38.2	3.5
YOLO-Lite ([18])	224	-	0.6M	1.0G	-	12.26	36
YOLOv3-tiny ([35])	320	Tiny Darknet	8.85M	3.3G	29	14	14
YOLOv4-tiny ([2])	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.

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	12
Ours	8.04M	8.09×	51.4	11.5

Table 3: Comparison of different pruning schemes.

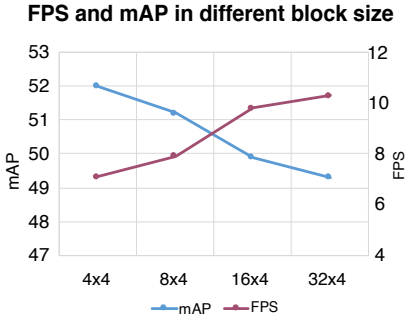

Figure 5: Accuracy (mAP) and speed (FPS) of different block size pruning results.

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 4 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.

4.3 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 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 5, 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.

5 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

[1] Alibaba. `https://github.com/alibaba/MNN`, 2020.

[2] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934*, 2020.

[3] Emmanuel J Candes, Michael B Wakin, and Stephen P Boyd. Enhancing sparsity by reweighted ℓ_1 minimization. *Journal of Fourier analysis and applications*, 14(5-6):877–905, 2008.

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: An automated end-to-end optimizing compiler for deep learning. In *the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[5] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[6] Xiaoliang Dai, Hongxu Yin, and Niraj K Jha. Nest: A neural network synthesis tool based on a grow-and-prune paradigm. *IEEE Transactions on Computers*, 68(10):1487–1497, 2019.

[7] Kaiwen Duan, Song Bai, Lingxi Xie, Honggang Qi, Qingming Huang, and Qi Tian. Centernet: Keypoint triplets for object detection. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 6569–6578, 2019.

[8] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *The International Conference on Learning Representations (ICLR)*, 2018.

[9] Ross Girshick. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448, 2015.

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

[11] Google. `https://www.tensorflow.org/lite`, 2020.

[12] Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic network surgery for efficient dnns. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2016.

[13] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems (NeurIPS)*, 2015.

[14] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969, 2017.

[15] Yang He, Ping Liu, Ziwei Wang, Zhilan Hu, and Yi Yang. Filter pruning via geometric median for deep convolutional neural networks acceleration. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[16] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018.

[17] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2017.

[18] Rachel Huang, Jonathan Pedoeem, and Cuixian Chen. Yolo-lite: a real-time object detection algorithm optimized for non-gpu computers. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 2503–2510. IEEE, 2018.

[19] Hei Law, Yun Teng, Olga Russakovsky, and Jia Deng. Cornernet-lite: Efficient keypoint based object detection. *arXiv preprint arXiv:1904.08900*, 2019.

[20] Tuanhui Li, Baoyuan Wu, Yujiu Yang, Yanbo Fan, Yong Zhang, and Wei Liu. Compressing convolutional neural networks via factorized convolutional filters. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[21] Yuxi Li, Jiuwei Li, Weiyao Lin, and Jianguo Li. Tiny-dsod: Lightweight object detection for resource-restricted usages. *arXiv preprint arXiv:1807.11013*, 2018.

[22] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*, pages 2980–2988, 2017.

[23] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[24] Li Liu, Wanli Ouyang, Xiaogang Wang, Paul Fieguth, Jie Chen, Xinwang Liu, and Matti Pietikäinen. Deep learning for generic object detection: A survey. *International journal of computer vision*, 128(2):261–318, 2020.

[25] Ning Liu, Xiaolong Ma, Zhiyuan Xu, Yanzhi Wang, Jian Tang, and Jieping Ye. Autocompress: An automatic dnn structured pruning framework for ultra-high compression rates. In *AAAI*, 2020.

[26] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: Single shot multibox detector. In *ECCV*, 2016.

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

[28] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. Rethinking the value of network pruning. In *International Conference on Learning Representations*, 2018.

[29] Xiaolong Ma, Fu-Ming Guo, Wei Niu, Xue Lin, Jian Tang, Kaisheng Ma, Bin Ren, and Yanzhi Wang. 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)*, 2020.

[30] Xiaolong Ma, Geng Yuan, Sheng Lin, Caiwen Ding, Fuxun Yu, Tao Liu, Wujie Wen, Xiang Chen, and Yanzhi Wang. Tiny but accurate: A pruned, quantized and optimized memristor crossbar framework for ultra efficient dnn implementation. In *ASP-DAC*, 2020.

[31] Chuhan Min, Aosen Wang, Yiran Chen, Wenyao Xu, and Xin Chen. 2pfpce: Two-phase filter pruning based on conditional entropy. *arXiv preprint arXiv:1809.02220*, 2018.

[32] Wei Niu, Xiaolong Ma, Sheng Lin, Shihao Wang, Xuehai Qian, Xue Lin, Yanzhi Wang, and Bin Ren. 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)*, 2020.

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

[34] Joseph Redmon and Ali Farhadi. Yolo9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7263–7271, 2017.

[35] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[36] Ao Ren, Tianyun Zhang, Shaokai Ye, Jiayu Li, Wenyao Xu, Xuehai Qian, Xue Lin, and Yanzhi Wang. Admm-nn: An algorithm-hardware co-design 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*, pages 925–938, 2019.

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

[38] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[39] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, Jun 2018.

[40] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2016.

[41] Ruichi Yu, Ang Li, Chun-Fu Chen, Jui-Hsin Lai, Vlad I Morariu, Xintong Han, Mingfei Gao, Ching-Yung Lin, and Larry S Davis. Nisp: Pruning networks using neuron importance score propagation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[42] Tianyun Zhang, Shaokai Ye, Kaiqi Zhang, Jian Tang, Wujie Wen, Makan Fardad, and Yanzhi Wang. A systematic dnn weight pruning framework using alternating direction method of multipliers. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018.

[43] Chenglong Zhao, Bingbing Ni, Jian Zhang, Qiwei Zhao, Wenjun Zhang, and Qi Tian. Variational convolutional neural network pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[44] Xiaotian Zhu, Wengang Zhou, and Houqiang Li. Improving deep neural network sparsity through decorrelation regularization. In *Ijcai*, pages 3264–3270, 2018.

[45] Zhuangwei Zhuang, Mingkui Tan, Bohan Zhuang, Jing Liu, Yong Guo, Qingyao Wu, Junzhou Huang, and Jinhui Zhu. Discrimination-aware channel pruning for deep neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.