Neural Pruning Search for Real-Time Object Detection of Autonomous Vehicles

[1]Pu Zhao, [1]Geng Yuan, [1]Yuxuan Cai, [2]Wei Niu, [3]Qi Liu, [3]Wujie Wen, [2]Bin Ren, [1]Yanzhi Wang, [1]Xue Lin
[1]*Northeastern University, Boston, MA*
[2]*College of William & Mary, Williamsburg, VA*
[3]*Lehigh university, PA*
{zhao.pu, yuan.geng, cai.yuxu}@northeastern.edu, wniu@email.wm.edu,
{qil219, wuw219}@lehigh.edu, bren@cs.wm.edu, {yanz.wang, xue.lin}@northeastern.edu

***Abstract*—Object detection plays an important role in self-driving cars for security development. However, mobile systems on self-driving cars with limited computation resources lead to difficulties for object detection. To facilitate this, we propose a compiler-aware neural pruning search framework to achieve high-speed inference on autonomous vehicles for 2D and 3D object detection. The framework automatically searches the pruning scheme and rate for each layer to find a best-suited pruning for optimizing detection accuracy and speed performance under compiler optimization. Our experiments demonstrate that for the first time, the proposed method achieves (close-to) real-time, 55ms and 97ms inference times for YOLOv4 based 2D object detection and PointPillars based 3D detection, respectively, on an off-the-shelf mobile phone with minor (or no) accuracy loss.**

***Index Terms*—real-time, object detection, autonomous driving**

I. Introduction

As the rapid development of the autonomous vehicles, which attempts to navigate roadways without human intervention, the environment sensing (known as perception) serves as a fundamental building block. Among various subtasks of perception, the object detection including 2D and 3D detection is one of the most important prerequisites to autonomous navigation. 2D and 3D detection makes use of the 2D image and 3D point clouds information from camera and LiDAR sensors, respectively, to detect objects in the environments, thus providing the autonomous navigation with the desirable knowledge about its environment and enabling high-level navigation computations and optimizations.

It is essential to implement real-time object detection for both 2D and 3D detection on autonomous vehicles as they need to interact with the environment instantaneously to avoid potential accidents. However, as 2D and 3D object detections are implemented with deep neural networks (DNNs) such as YOLO [1] and PointPillars [2], respectively, with tremendous memory and computation requirements, it is challenging to achieve real-time on autonomous vehicles with limited memory and computation resources. The more powerful high-end GPUs are usually too costly and power-hungry to be deployed on vehicles. Thus, it is desirable to satisfy the real-time requirement within the limited memory and computation constraints for detection models.

DNN weight pruning [3], [4] has been proved as an effective model compression technique to remove redundant weights, thereby reducing computations and accelerating inference. Existing work mainly focus on *unstructured pruning* [4]–[6] where arbitrary weight can be removed, and (coarse-grained) *structured pruning* [3], [6]–[10] to eliminate whole filters. The former results in high accuracy but limited hardware parallelism (and acceleration), while the latter is the opposite.

Recent work [11]–[13] proposed the novel concept of *fine-grained structured pruning* scheme: *Pattern-based pruning* [11], [12] assigns potentially different patterns to convolutional (CONV) kernels, while *block-based pruning* [13] divides the weight matrix into equal-sized blocks and performs independent row/column pruning in each block. High accuracy can be achieved as a result of intra-kernel (or intra-block) flexibility, while high hardware parallelism (and mobile inference acceleration) can be achieved with the assist of compiler-level code generation techniques [12]. Despite the promising results, pattern-based pruning [11], [12] is only applied to 3×3 CONV layers, while block-based pruning [13] is only suitable for fully-connected (FC) layers, which limit their applicability.

As the **first contribution**, we generalize the concept of fine-grained structured pruning to cover all types of CONV and FC layers, which is necessary for both 2D and 3D real-time object detection of autodriving. In this way, we can fully leverage the new opportunity to achieve high accuracy and high acceleration simultaneously *with the aid of advanced compiler optimizations*. Block-based pruning scheme can be naturally extended to 1×1 CONV layers. We extend pattern-based pruning to larger kernel sizes beyond 3×3, overcoming the limitation of increased computation overhead with an increasing number of pattern types. We develop a comprehensive, compiler-based automatic code generation framework *supporting different (proposed and existing) pruning schemes in a unified manner*, and *different rates for different layers*.

While our compiler optimizations provide notable mobile acceleration and support of various sparsity schemes, it introduces *a larger model optimization space* beyond the scope of prior work: Different sparsity schemes result in different accuracy performances, and different accelerations under compiler optimizations. Motivated by the recent idea of Neural Architecture Search (NAS) [14], [15], we propose to perform a novel *compiler-aware neural pruning search*, automatically determining the pruning scheme and rate (including bypass) for each individual layer. The *objective* is to maximize ac-

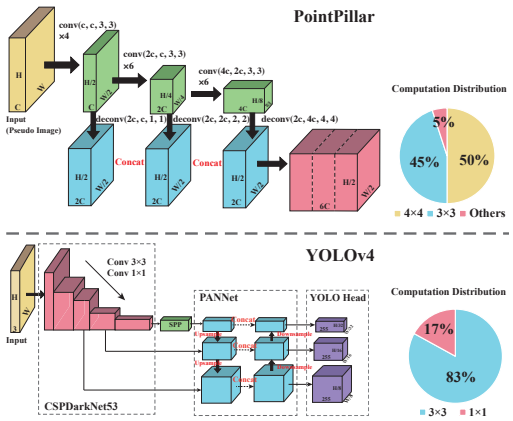

Fig. 1. The PointPillars data flow for 3D detection and YOLOv4 data flow for 2D object detection with computation distributions.

curacy satisfying an inference latency constraint on the target mobile device. The DNN latency will be actually measured on target device, thanks to the fast auto-tuning capability of our compiler for efficient inference on different mobile devices.

Our framework employs pre-trained DNN models as starting point, which are already optimized for high detection accuracy in autodriving tasks to accelerate the pruning search. The overall latency constraint is satisfied through the synergic efforts of (i) incorporating overall DNN latency constraint into automatic search process, and (ii) effective pruning algorithm to perform weight training/pruning accordingly. To perform efficient search under a large search space, we propose a meta-modeling procedure with Bayesian optimization (BO). We can achieve (close-to) real-time, 55ms and 97ms inference times for YOLOv4 based 2D detection and PointPillars based 3D detection, respectively, on a mobile phone with minor (or no) accuracy loss. Our method on 2D detection outperforms other acceleration frameworks such as TVM [16] and MNN [17], while we are the first to support 3D detection on mobile.

II. Background and Related Work

2D object detection detects various objects including pedestrians, cyclists and cars from 2D camera images. Similarly, 3D object detection detects objects in the environment from 3D LiDAR images with point clouds. They are crucial for autodriving perception to gain basic knowledge about its environment and almost all of the navigation decisions are built on the detection information. YOLOv4 [1] and PointPillars [2] are two popular DNN models for 2D and 3D object detection, respectively. YOLOv4 has a backbone and a head to detect and regress 2D boxes. Similarly, a PointPillar model consists of three main stages: (1) A feature encoder network to convert a point cloud to a sparse pseudo-image; (2) a 2D CONV backbone to process the pseudo-image into high-level representation; and (3) a detection head to regress 3D boxes. We show the data flow and computation distribution for YOLOv4 and PointPillars in Fig. 1. We observe that (1) there are various kinds of CONV operations in the model such as 1×1, 3×3 and 4×4, and (2) the CONV operations take up most of the computations in models.

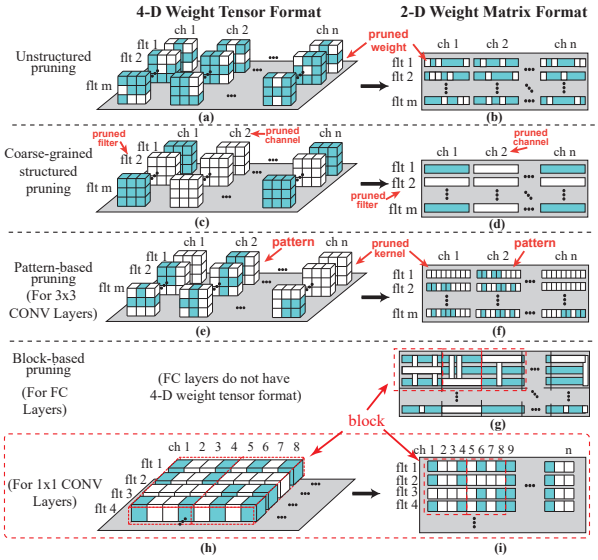

Fig. 2. Different weight pruning schemes for CONV layers using 4D tensor and 2D matrix representation.

A. Weight Pruning: Schemes and Algorithms

Existing weight pruning research can be categorized according to pruning schemes and pruning algorithms.

Pruning Scheme: Previous weight pruning work can be categorized according to the pruning scheme: *unstructured pruning* [4], [5], [18], *coarse-grained structured pruning* [3], [6]–[8], and fine-grained structured pruning schemes including *pattern-based* [11], [12] and *block-based* [13] pruning.

Unstructured pruning (Fig. 2 (a) and (b)) removes weights at arbitrary position. Despite the large compression rate, the irregular sparse weight matrix with indices damages the parallel implementations, leading to limited acceleration on hardware.

To overcome this, many work [3], [7]–[9] studied coarse-grained structured pruning at the level of filters as shown in Fig. 2 (c) and (d). With the elimination of filters, the pruned model still maintains the network structure with high regularity which can be parallelized on hardware. The downside is the obvious accuracy degradation, limiting compression rate.

Fig. 2 (e) and (f) show pattern-based pruning [11], [12] and block-based pruning [13], respectively, as representative fine-grained structured pruning schemes. The former is applied to 3×3 CONV layers, and the latter to FC layers, in the original papers. Pattern-based pruning assigns a pattern (from a predefined library) to each CONV kernel, maintaining a fixed number of weights in each kernel. As shown in the figure, each kernel reserves 4 non-zero weights (on a pattern) out of the original 3×3 kernels. Besides being assigned a pattern, a kernel can be completely removed to achieve higher compression rate. On the other hand, block-based pruning divides the FC weight matrix into equal-sized blocks and performs independent row/column pruning in each block.

For these fine-grained structured pruning schemes, high accuracy can be achieved as a result of intra-kernel (or intra-block) flexibility, while high hardware parallelism (and mobile acceleration) can be achieved with the assist of compiler-based

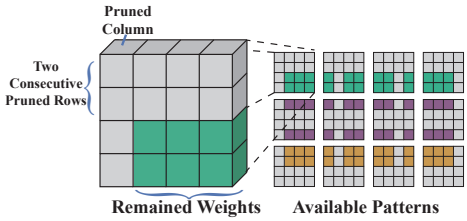

Fig. 3. Example of patterns on 4×4 CONV layer.

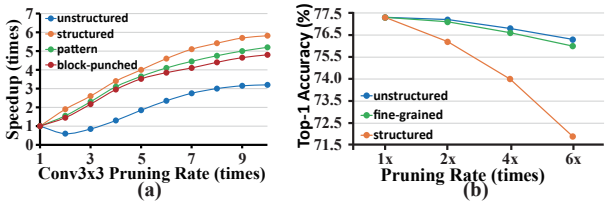

Fig. 4. (a) Speedup vs. pruning rate with different pruning schemes, (b) accuracy comparison of pruning schemes under different pruning rate.

code generation. For pattern-based pruning [12], compiler can perform filter reorder to group kernels with the same pattern, and allocate a thread to each group, thereby achieving high hardware parallelism. For block-based pruning [13], thanks to the large weight matrices, the remaining computation in each block still achieves high parallelism on mobile CPU/GPU with computation overhead eliminated by compiler.

Our goal is to overcome the limited application of pattern-based and block-based pruning, and generalize to all types of CONV and FC layers. 1×1 CONV layers do not exhibit intra-kernel flexibility and thus are not suitable for pattern-based pruning, but we found that block-based pruning can be naturally extended to 1×1 layers because such CONV layers are also computed as matrix multiplication [19], [20]. On the other hand, there is a **key difficulty** in generalizing pattern-based pruning to larger kernel sizes beyond 3×3: It results in a large number of pattern types, which incurs notable computation overhead in compiler-generated executable codes.

Pruning Algorithm: Two main categories exist: *heuristic pruning algorithm* [4], [5], [8], [9] and *regularization-based pruning algorithm* [3], [6], [7], [21]. Heuristic pruning was firstly performed in an iterative, magnitude-based manner on unstructured pruning [5], and gets improved in later work [4] and incorporated into coarse-grained structured pruning [8], [9], [22]. Regularization-based algorithm uses mathematics-oriented method to deal with pruning problem. Early work [3], [7] incorporates ℓ_1 or ℓ_2 regularization in loss function to solve filter/channel pruning problems. In [21], an advanced optimization solution framework ADMM (Alternating Direction Methods of Multipliers) is utilized to achieve dynamic regularization penalty which notably reduces accuracy loss.

III. Extending Pattern-based Pruning to Large Kernel Sizes

There are two requirements in order for pattern-based pruning to deliver its promise in hardware acceleration with the assist of compiler code generation. First, different patterns should have the same number of remaining weights (and computations) to achieve load balancing among threads at compiler level. This is satisfied in prior work on 3×3 kernels (only focusing on 4-entry patterns) and can be well generalized to larger kernel sizes. Second and more importantly, the number of pattern types needs to be restricted to below 20 to limit the computation overhead in compiler-generated executable codes. This can be a major difficulty for larger kernel sizes beyond 3×3. Using 4×4 kernels as an example that is necessary in 3D object detection, and consider only 6-entry patterns. Then there are $\binom{16}{6}$ pattern types if no restriction is enforced, which is much beyond the capability of compiler.

To overcome this difficulty, we propose a systematic way of generalizing pattern-based pruning with restriction on pattern types. We use 4×4 kernels as example, but can be generalized to larger kernels like 5×5 or 7×7. As Fig. 3 shows, we limit to intra-kernel row and column pruning in each kernel for defining patterns, further restricting to removing two consecutive rows and one column. In this way, each pattern has 6 remaining weights free for training, and there are 12 patterns in total. Compiler code generation uses filter/channel reorder similar to [12], with tolerable computation overhead thanks to the limited number of pattern types.

Compiler Optimizations: We develop a comprehensive, compiler-based automatic code generation framework supporting the pattern-based pruning schemes for various CONV kernel sizes in a unified manner. It also supports other pruning schemes such as unstructured, coarse-grained structured, block-based pruning. Fast *auto-tuning* capability is incorporated for efficient end-to-end inference on different mobile CPU/GPU. We achieve superior inference acceleration on both dense (before pruning) and sparse DNN models, as to be shown in the experimental results.

IV. Motivation of Neural Pruning Search

A key **observation** is that different sparsity schemes (pattern-based, block-based, coarse-grained, etc.) have different accuracy and acceleration performances under compiler optimizations (when computation (MACs) is the same).

Different Pruning Schemes: Fig. 4 (a) shows the computation speedup vs. pruning rate of a 3×3 CONV layer with different pruning schemes, measured on mobile CPU (Qualcomm Snapdragon 865 Octa-core CPU) of a Samsung Galaxy S20 phone. We choose the input feature map size of 56×56 and 256 input and output channels. We can observe that, with compiler optimizations, pattern-based pruning consistently outperforms the unstructured pruning and achieves comparable acceleration as coarse-grained structured pruning below 5× pruning. Since, under reasonable pruning rate of fine-grained structured pruning schemes, the remaining weights in each layer are still sufficient to fully utilize hardware parallelism.

Fig. 4 (b) shows the Top-1 accuracy performance of ResNet-50 with different pruning schemes, applying uniform pruning rate to all CONV layers (ADMM-based pruning algorithm). The fine-grained pruning case applies pattern-based pruning to 3×3 CONV layers and block-based pruning to 1×1 CONV layers. We can observe that fine-grained pruning can preserve high accuracy, only slightly lower than unstructured pruning.

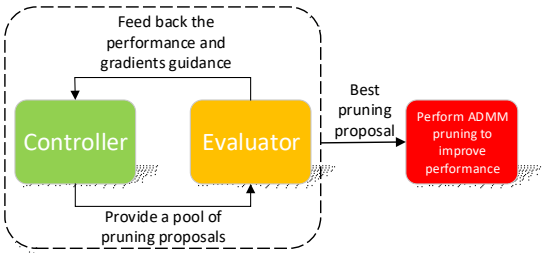

Fig. 5. Automatic network pruning search framework

Impact of Number of Layers: The number of computation layers is another critical factor that affects inference latency. To show the impact, we make a narrower-but-deeper version of ResNet-50 by doubling the number of layers, while keeping computation MACs the same as the original ResNet-50. And the inference speed of the narrower-but-deeper version is 1.22× slower than the original one using mobile GPU (44ms vs. 36ms). This is because a larger number of layers introduce more intermediate results and hence more frequent data access to the main memory. And mobile CPU/GPU cannot be fully utilized due to a large number of memory-intensive layers.

Based on the above observations, it is desirable to perform a compiler-aware neural pruning search, automatically determining the *pruning scheme and rate* for each individual layer. The objective is to *maximize DNN accuracy satisfying an inference latency constraint* when actually executing on the target mobile device, accounting for compiler optimizations.

V. Automatic Neural Pruning Search

As shown in Fig. 5, the framework consists of two basic components: a *controller* and an *evaluator*. The controller first generates various *pruning proposals* from the search space. Then the evaluator evaluates their detection accuracy and speed performance. Based on the performance, the evaluator provides guidance for controller about what a satisfying pruning proposal looks like. Next the controller generates new pruning proposals with the guidance. After iterations, the controller outputs the best pruning proposal with desirable detection performance while satisfying the real-time requirement.

With the derived best pruning proposal, it achieves real-time inference with compiler optimization. Besides the satisfying speed performance, to further improve detection performance, we choose to adopt ADMM pruning [21] to perform an enhanced pruning following the best proposal, after comparing multiple pruning algorithms. It supports different sparsity schemes with the help of group-Lasso regularization [3].

A. Controller

The controller generates *pruning proposals* from the search space. Each pruning proposal is a directed graph consisting of the pruning scheme and pruning rate for each layer of the model. For example, it has 10 nodes for a 5-layer DNN model.

1) Search Space: Each pruning proposal contains layer-wise pruning scheme and pruning rate, as shown in Tab. I.

Per-layer pruning schemes: The controller can choose from filter (channel) pruning [23], pattern-based pruning (including our extension to larger kernel sizes) and block-based pruning [13] for each layer. As different layers may have different best-suited pruning schemes, we allow the controller the flexibility to choose different pruning schemes for different layers, also supported by our compiler code generation.

TABLE I
Search space for each DNN layer

Pruning scheme	{Filter [23], Pattern-based, Block-based [13]}
Pruning rate	{ 1×, 2×, 2.5×, 3×, 5×, 7×, 10×, skip }

Per-layer pruning rate: We can choose from the list $\{1\times, 2\times, 2.5\times, 3\times, 5\times, 7\times, 10\times, \text{skip}\}$, where $1\times$ means the layer is not pruned, and "skip" means bypassing this layer.

2) Pruning Proposal Updating: The evaluator provides the *gradients guidance* specified in Sec. V-B2 to guide the controller for generating new pruning proposals. The gradients guidance contains a currently best pruning proposal and its corresponding replacement probability obtained from its gradients. The controller determines whether to replace each node in the proposal according to the replacement probability. Next if replaced, the controller chooses randomly from two nodes with the lowest probabilities as its replacement.

B. Evaluator

The evaluator needs to evaluate pruning proposal performance. We define the performance measurement (reward) as:

$$r = V - \alpha \cdot \max(0, t - T), \tag{1}$$

where V is the validation mean average precision (mAP) of the model, t is the model inference speed or latency, which is actually measured on a mobile device[1]. T is the latency requirement threshold. Generally, r is high when it satisfies real-time requirement ($t < T$) with high mAP. Otherwise r is small if the real-time latency requirement is violated.

1) Evaluation with BO: As evaluating each proposal needs to prune and retrain the model with multiple epochs, incurring large time cost, we use Bayesian optimization (BO) [24] to accelerate evaluation. The controller generates a *proposal pool* with K pruning proposals. We first use BO to select B ($B < K$) proposals with potentially better performance. Then the selected proposals are evaluated to derive the accurate detection and speed performance while the rest $(K - B)$ potentially weak proposals are not evaluated. Thus, we reduce the number of actual evaluated proposals.

To deal with the graph-like pruning proposals, we build a Gaussian process (GP) for BO with a Weisfeiler-Lehman (WL) graph kernel [25]. We select proposals according to the acquisition function values (we employ *Expected Improvement* [26]). Algorithm 1 provides a summary.

We illustrate the WL graph kernel in Fig. 6. More specifically, the WL kernel compares two directed graphs in iterations. In the m-th WL iteration, it first obtains the graph feature vectors $\phi_m(g)$ and $\phi_m(g')$ for two graphs. Then it compares the two graphs with $k_{\text{base}}\big(\phi_m(g), \phi_m(g')\big)$ where we employ

[1]The compiler code generation is much faster than DNN training thanks to the auto-tuning capability, and can be performed in parallel with the pruning as measuring inference speed does not need accurate weight values.

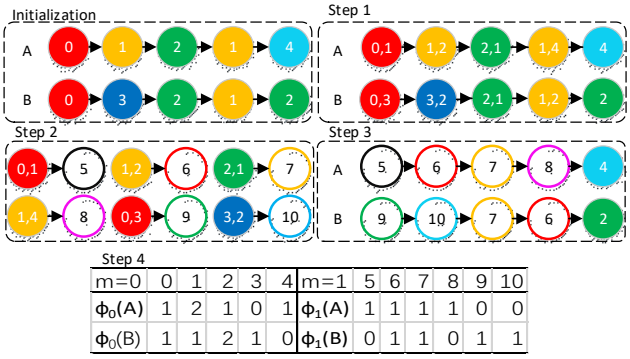

Fig. 6. WL kernel illustration. At initialization, there are two pruning proposals with features at $m = 0$. At step 1, WL kernel collects the next label of each node. At step 2, it re-labels the new nodes with neighbour information. At step 3, it obtains a new graph with features at $m = 1$. At step 4, WL kernel compares the histogram on both $m = 0$ and $m = 1$ features. The iteration repeats until $m = M$.

dot product as k_{base} here. The iterative procedure stops until $m = M$ and the resultant WL kernel is

$$k^M_{\text{WL}}(g, g') = \sum_{m=0}^{M} w_m k_{\text{base}}\big(\phi_m(g), \phi_m(g')\big). \tag{2}$$

where w_m contains the weights for each WL iteration m, which is set to equal for all m following [25].

After selecting B pruning proposals from the pool, we evaluate their performance using magnitude based framework [5] following their pruning proposals for each layer.

2) Gradients Guidance: To guide the proposal updating, we employ the derivatives of the GP predictive mean. Formally, the derivative with respect to the j-th element of $\phi^t = \phi(G_t)$ is also Gaussian with an expected value:

$$\mathbb{E}_{p(r|G_t,\mathcal{D}_{t-1})}\left[\frac{\partial r}{\partial[\phi^t]_j}\right] = \frac{\partial \mu}{\partial[\phi^t]_j} = \frac{\partial\langle \phi^t, \Phi^{1:t-1}\rangle}{\partial[\phi^t]_j} L^{-1}_{1:t-1}\mathbf{r}_{1:t-1} \tag{3}$$

where $\Phi_{1:t-1}$, $L_{1:t-1}$ and $\mathbf{r}_{1:t-1}$ are the stacked feature matrix, kernel values and rewards from previous observations.

Following Eq. (3), we can obtain $\mathbb{E}\left[\frac{\partial r}{\partial[\phi^t_0]_j}\right]$ where ϕ^t_0 is the list of the number of each node. Positive gradients show that the node is beneficial to improve the reward, while negative gradients mean that the node decreases the performance and it should be replaced. To make the gradients more illustrative, we transform the gradients into a probability distribution (replacement probability) using a sigmoid transformation on the negative of the gradients and then normalize them. Thus, negative gradients lead to high replacement probabilities.

To summarize, the evaluator provides the gradient guidance including the best evaluated pruning proposal and its corresponding replacement probability obtained from its gradients.

VI. Experimental Results

A. Experiment Setup

For 2D object detection, we use a YOLOv4 [1] model as starting point and test on COCO dataset [27]. For 3D detection, we employ the PointPillars as starting point [2] and test on KITTI dataset [28]. We use 50 GPUs for parallel training and pruning search and it takes about 12 days to find the best pruning proposal in each experiment. In Eq. (1), we set α to 0.01 and the mobile inference time is measured in milliseconds. We set $w_m = 1$ in Eq. (2). The pool size K is set to 50 and the Bayesian batch size B is set to 10. All the acceleration results are tested on the mobile GPU (Qualcomm Adreno 640) of a Samsung Galaxy S20 smartphone.

Algorithm 1 Evaluation with BO

Input: Data $\mathcal{D}$, BO batch size B, BO acquisition function $\alpha(\cdot)$
Output: The best pruning proposal g
for steps **do**
  Generate a pool of candidate pruning proposals $\mathcal{G}_c$;
  Select $\{\hat{g}^i\}_{i=1}^B = \arg\max_{g\in\mathcal{G}_c} \alpha(g|\mathcal{D})$;
  Evaluate the proposal and obtain reward $\{r^i\}_{i=1}^B$ of $\{\hat{g}^i\}_{i=1}^B$;
  Obtain the gradients guidance information;
  $\mathcal{D} \leftarrow \mathcal{D} \cup (\{\hat{g}^i\}_{i=1}^B, \{r^i\}_{i=1}^B)$;
  Update GP of BO with $\mathcal{D}$;
end for

TABLE II
Comparison of various pruning methods on YOLOv4

Pruning Methods	Parameter #	Computation # (MACs)	mAP	Mobile GPU Speed (ms)
Original	64.36M	17.9G	56.5	285.7
Pattern [11]	16.09M	5.58G	47.6	114.9
Filter [29]	4.33M	1.87G	25.2	47.6
Unstructured [21]	4.33M	1.87G	49.9	98.6
Ours	4.33M	1.87G	49.3	55.2

B. Performance on 2D Object Detection

As shown in Table II, we applied our proposed method to YOLOv4 and compared it with other pruning methods in terms of accuracy (mAP) and end-to-end inference latency on mobile GPU, with our compiler framework. Our method reduces the original computation from 17.9G MACs to 1.87G MACs and achieves 5.18× inference acceleration (285.7ms vs. 55.2ms). We also adopt the same pruning rate to YOLOv4 using other pruning schemes except pattern-based pruning, because there are 17% weights that come from 1×1 CONV layer cannot be pruned using pattern-based pruning and hence a lower overall pruning rate. Under the same pruning rate, our method is 16% slower than structured pruning on mobile GPU but achieves much higher accuracy (49.3 vs. 25.2 in mAP). With slightly lower accuracy, our method is 1.79× faster than unstructured pruning. And our method outperforms the pattern-based pruning in both accuracy and inference speed.

Fig. 7 shows the accuracy vs. latency comparison results of our method with other DNN inference frameworks and using other object detectors. The rectangular shapes show the comparison results of dense (unpruned) YOLOv4, as general sparsity is not supported in other frameworks. By leveraging our compiler optimizations, our method outperforms other representative DNN inference frameworks, including TensorFlow-Lite, TVM, and MNN. All the red star shapes represent our pruned models with different latency constraints and using YOLOv4 as the starting point. And our method achieves near Pareto-optimality considering both accuracy and latency. The results clearly show the effectiveness of our proposed method.

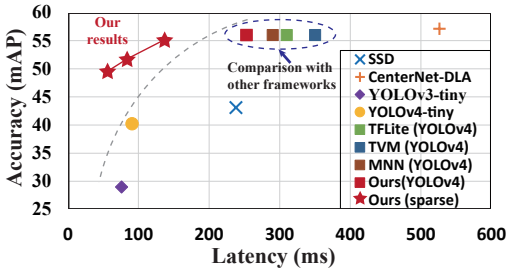

Fig. 7. mAP vs. latency for various object detection approaches.

TABLE III
COMPARISON OF VARIOUS PRUNING METHODS FOR POINTPILLARS

Methods (grid size)	Para. #	Comp. # (MACs)	Speed (ms)	Car 3D detection		
				Easy	Moderate	Hard
PointPillars (0.16)	5.8M	60G	542	84.99	74.11	69.53
Ours (0.16)	1.1M	10.7G	189	**85.50**	**76.58**	**70.23**
PointPillars (0.24)	5.8M	28G	257	84.05	74.99	68.30
Filter [29] (0.24)	0.8M	4.0G	81	81.54	68.10	65.90
Pattern [11] (0.24)	0.8M	3.9G	111	80.97	73.77	68.05
Ours (0.24)	0.8M	3.9G	97	**85.20**	**75.57**	**68.37**

C. Performance on 3D Object Detection

We show the performance of the original unpruned PointPillars model and the model derived by our method with different grid sizes (0.16m and 0.24m, where large grid size means small input size for the model) in Tab. III. The real-time requirements are set to 200ms for 0.16m and 100ms for 0.24m. We can observe that increasing grid size reduces the pseudo-image input size, leading to smaller parameter and computation numbers, and faster inference speed on mobile GPUs. For the same grid size, our method can significantly reduce the parameter count and computation, thus satisfying the real-time requirement with state-of-the-art detection performance.

For a grid size of 0.24m, the pruning ratio of other pruning schemes are set to the same with the overall pruning ratio of our pruned model (86%). As observed, the proposed method can achieve the best detection performance compared with other methods where all the layers share the same pruning scheme, demonstrating the advantages of flexible pruning scheme for each layer. Besides with compiler optimization, filter pruning is the fastest but suffers from obvious detection performance degradation.

The proposed method can process one LiDAR image within 97ms with the highest precision, demonstrating the superior performance of the proposed method to achieve (close-to) real-time inference on mobile with state-of-the-art detection performance. Although other DNN inference frameworks exist such as Tensorflow-Lite [19], TVM [16] and MNN [17], we are the **first** to support 3D object detection on mobile device.

VII. CONCLUSION

We propose neural pruning search to achieve real-time 2D and 3D object detection on mobile for autonomous vehicles. Our experiments demonstrate that for the first time, the proposed method achieves (close-to) real-time, 55ms and 97ms inference times for YOLOv4 based 2D detection and PointPillars based 3D detection, respectively, on an off-the-shelf mobile phone with minor (or no) accuracy loss.

VIII. ACKNOWLEDGEMENTS

This project is partly supported by National Science Foundation CNS-1932351, CNS-1739748, CNS-1909172, and CCF-2006748, Army Research Office/Army Research Laboratory (ARO) W911NF-20-1-0167 (YIP) to Northeastern University, a grant from Semiconductor Research Corporation (SRC), and Jeffress Trust Awards in Interdisciplinary Research. Any opinions, findings, and conclusions or recommendations in this material are those of the authors and do not necessarily reflect the views of NSF, ARO, SRC, or Thomas F. and Kate Miller Jeffress Memorial Trust.

REFERENCES

[1] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "Yolov4: Optimal speed and accuracy of object detection," *arXiv:2004.10934*, 2020.
[2] A. H. Lang, S. Vora *et al.*, "Pointpillars: Fast encoders for object detection from point clouds," in *CVPR*, 2019, pp. 12 697–12 705.
[3] W. Wen, C. Wu *et al.*, "Learning structured sparsity in deep neural networks," in *NeurIPS*, 2016, pp. 2074–2082.
[4] Y. Guo, A. Yao, and Y. Chen, "Dynamic network surgery for efficient dnns," in *NeurIPS*, 2016, pp. 1379–1387.
[5] S. Han, J. Pool *et al.*, "Learning both weights and connections for efficient neural network," in *NeurIPS*, 2015, pp. 1135–1143.
[6] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell, "Rethinking the value of network pruning," *arXiv preprint arXiv:1810.05270*, 2018.
[7] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *ICCV*, 2017, pp. 1389–1397.
[8] J.-H. Luo, J. Wu, and W. Lin, "Thinet: A filter level pruning method for deep neural network compression," in *ICCV*, 2017, pp. 5058–5066.
[9] R. Yu, A. Li *et al.*, "Nisp: Pruning networks using neuron importance score propagation," in *CVPR*, 2018, pp. 9194–9203.
[10] N. Liu, X. Ma *et al.*, "Autocompress: An automatic dnn structured pruning framework for ultra-high compression rates," in *AAAI*, 2020.
[11] X. Ma *et al.*, "Pconv: The missing but desirable sparsity in dnn weight pruning for real-time execution on mobile devices," in *AAAI*, 2020.
[12] W. Niu *et al.*, "Patdnn: Achieving real-time dnn execution on mobile devices with pattern-based weight pruning," *arXiv:2001.00138*, 2020.
[13] P. Dong, S. Wang *et al.*, "Rtmobile: Beyond real-time mobile acceleration of rnns for speech recognition," *arXiv:2002.11474*, 2020.
[14] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," in *ICLR*, 2017.
[15] H. Liu, K. Simonyan, and Y. Yang, "Darts: Differentiable architecture search," *arXiv preprint arXiv:1806.09055*, 2018.
[16] T. Chen, T. Moreau *et al.*, "Tvm: An automated end-to-end optimizing compiler for deep learning," in *USENIX*, 2018, pp. 578–594.
[17] https://github.com/alibaba/MNN.
[18] H. Mao, S. Han *et al.*, "Exploring the regularity of sparse structure in convolutional neural networks," *arXiv:1705.08922*, 2017.
[19] https://www.tensorflow.org/mobile/tflite/.
[20] A. Paszke, S. Gross *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *NeurIPS*, 2019.
[21] T. Zhang, S. Ye *et al.*, "Systematic weight pruning of dnns using alternating direction method of multipliers," *ECCV*, 2018.
[22] X. Dong and Y. Yang, "Network pruning via transformable architecture search," in *NeurIPS*, 2019, pp. 759–770.
[23] Z. Zhuang, M. Tan *et al.*, "Discrimination-aware channel pruning for deep neural networks," in *NeurIPS*, 2018, pp. 875–886.
[24] Y. Chen, A. Huang *et al.*, "Bayesian optimization in alphago," *arXiv:1812.06855*, 2018.
[25] N. Shervashidze, P. Schweitzer *et al.*, "Weisfeiler-lehman graph kernels," *Journal of Machine Learning Research*, vol. 12, no. 77, 2011.
[26] C. Qin, D. Klabjan, and D. Russo, "Improving the expected improvement algorithm," in *NeurIPS*, 2017, pp. 5381–5391.
[27] T.-Y. Lin *et al.*, "Microsoft coco: Common objects in context," in *ECCV*.
[28] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the kitti vision benchmark suite," in *CVPR*, 2012.
[29] Y. He, P. Liu *et al.*, "Filter pruning via geometric median for deep convolutional neural networks acceleration," in *CVPR*, 2019.