Sparse PointPillars: Exploiting Sparsity in Birds-Eye-View Object Detection

Kyle Vedder and **Eric Eaton**
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104
{kvedder, eeaton}@seas.upenn.edu

Abstract

Bird's Eye View (BEV) is a popular representation for processing 3D point clouds, and by its nature is fundamentally sparse. Motivated by the computational limitations of mobile robot platforms, we take a fast high-performance BEV 3D object detector—PointPillars—and modify its backbone to exploit this sparsity, leading to decreased runtimes. We present preliminary results demonstrating decreased runtimes with either the same performance or a modest decrease in performance, which we anticipate will be remedied by model-specific hyperparameter tuning. Our work is a first step towards a new class of 3D object detectors that exploit sparsity throughout their *entire* pipeline in order to reduce runtime and resource usage while maintaining good detection performance.

1 Introduction and Related Work

Following the initial development effort of robots, when concerns about resource utilization and power are downplayed in favor of creating intelligent behavior, roboticists often go through a second phase of scaling the system down to fit within commercially viable limits of power, memory, and computation without significantly sacrificing performance. Many autonomous vehicle manufacturers are currently facing this challenge, and it is even more pronounced for developers of intelligent mobile robots, when resources are limited from the very start. For example, even a larger, high-end robot like the Fetch [19] is not able to support several desktop-grade GPUs plus high-end CPUs in order to run its control stack, forcing them to settle for smaller embedded systems like NVidia's Jetson Xavier[1] or Intel's NUC[2]. Smaller platforms such as quadcopters often struggle to handle the weight or power requirements of even these embedded systems. This motivates the problem of developing techniques that reduce the resource usage of existing machine learning models (e.g., object detectors) while preserving their performance — models need not just barely fit on smaller devices, but they need to do so while sharing the device's resources with other components.

One general-purpose solution to this problem, model quantization [14, 18, 22, 2], first trains ML models in the standard fashion using floating point weights and then, after training, converts some [2] or all [14] weights into integer [22] or binary [9] quantized values that are faster to multiply than floating point values. The resulting quantized network is then finetuned, resulting in approximately the same performance while running significantly faster on accelerators (e.g., GPUs [8]), low-end compute devices (e.g., mobile phones [20]), or specialized hardware (e.g., FPGAs [16]).

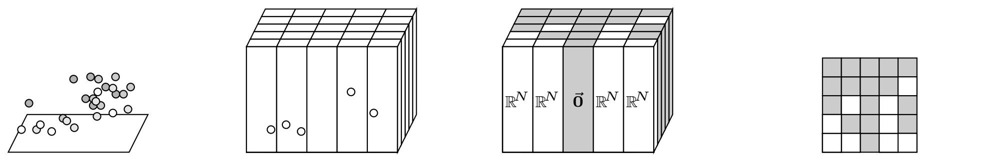
Architecture-specific approaches like model reparameterization address this problem by converting models into an efficient format. For example, ResNet [7] backbones are common in image recognition

[1] https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit
[2] https://www.intel.com/content/www/us/en/products/details/nuc.html

Preprint. Under review.

systems [13, 17], and their defining feature is the residual block which adds a convolved embedding to the input embedding. As a result, inference in ResNets is more involved than simple feed-forward networks like VGG [15], and thus adds memory, compute, and latency overhead. RepVGG [4] can reparameterize trained ResNets into simple VGG-style feed-forward networks while being mathematically equivalent, reducing memory usage and latency without changing model performance.

Other approaches use data representations that reduce the computational burden, such as exploiting input sparsity. For example, a common approach for 3D object detection in point clouds is to voxelize the 3D space and perform 3D convolutions through a pipeline similar to 2D object detection [23, 21, 1]. 3D convolution of these voxels dominates network runtime, but the voxels tend to be highly sparse; point clouds from the self-driving KITTI benchmark contain over 100,000 points, but when voxelized into 20cm cubes, over 90% of the voxels are empty [23]. This motivates sparsity-aware convolutional methods such as SECOND [21], which perform sparse 3D convolutions that significantly reduces model runtime without an impact to performance, and directly motivates our work.



(a) Raw Pointcloud (b) Pillarized Points (c) Vectorized Pillars (d) Sparse BEV Pseudoimage

Figure 1: PointPillars' pillarization and Bird's Eye View (BEV) pseudoimage creation process.

In this paper, we address this problem of reducing resource usage while preserving performance within PointPillars [10], a popular 3D object detector that operates on point clouds from a Bird's Eye View (BEV). PointPillars segregates the raw points (Figure 1a) into pillars (Figure 1b) and vectorizes these point collections into N dimensional vectors with empty pillars represented by the zero vector $\vec{0} = \{0\}^N$ (Figure 1c), resulting in a sparse BEV pseudoimage of the scene (Figure 1d). PointPillars then processes the pseudoimage with a dense 2D convolutional backbone (Appendix B, Figure 6a) and then predicts bounding boxes using a Single-Stage Detector (SSD) [12]. The PointPillars Backbone is the most computationally expensive component of the pipeline, but the pseudoimage it processes is highly sparse. As a result, there are large sections of the image that have no information but are still convolved by the backbone, leading to inefficiencies.

To eliminate these inefficiencies, **this paper proposes a modified pipeline for PointPillars that maintains and exploits end-to-end sparsity** to reduce runtime and resource usage while maintaining reasonable performance. Concretely, we propose two changes to PointPillars, highlighted in Figure 2:

1. A new Feature Net that maintains the pseudoimage's natural representation of a coordinate sparse tensor, removing the overhead of converting it to a dense tensor representation (Section 2.1), and
2. A new Backbone that exploits and preserves the natural sparsity of the pseudoimage via sparse [21] and submanifold [6] convolutions (Section 2.2).

We show empirically that these two changes result in roughly the same quality detections on the KITTI dataset while reducing the computational requirements and thus the runtime of the network. The code for the full models and their experimental configurations is publicly available[3].

[3] `https://github.com/kylevedder/SparsePointPillars`

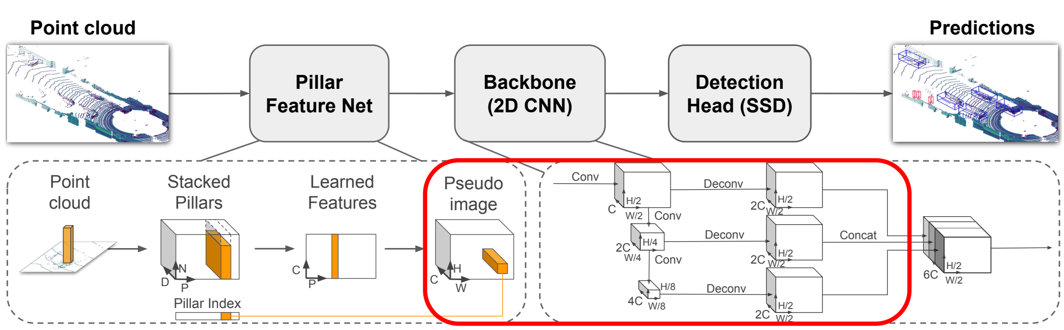


Figure 2: PointPillars [10] pipeline with our Sparse PointPillars's modified sections circled in red.

2 PointPillars Modifications for End-to-End Sparsity

This section describes our modifications to PointPillars in order to maintain and exploit sparsity end-to-end throughout the processing pipeline. We then empirically validate our approach in Section 3.

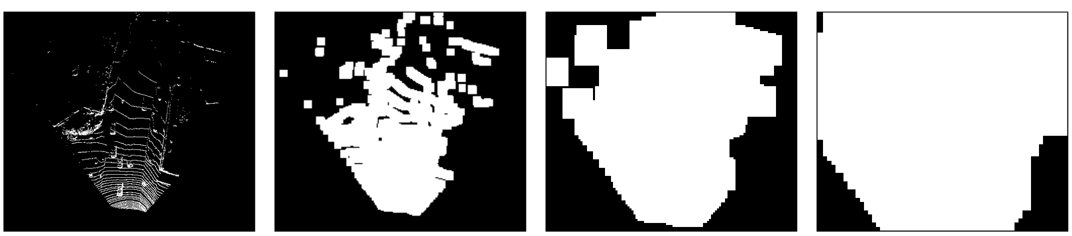
2.1 Replacement Pillar Feature Net

PointPillars' pillerization process (Figure 1) is implemented using a Feature Net that *gather*s the non-empty pillars of the full scene into a dense tensor representation with a fixed number of pillars and a fixed number of points per pillar (set a priori), along with the coordinate location of each pillar. The Feature Net then vectorizes each pillar using a PointNet-like vectorizer [3]. In the original PointPillars' Feature Net, these resulting vectors are then *scatter*ed back into a dense tensor in the shape of the full scene. In our modified PointPillar Feature Net, we replace this *scatter* step.

Instead of *scatter*ing back into a dense tensor, we construct a sparse tensor in the coordinate (COO) format[4] using the coordinate information already recorded during the *gather* step. This constant-time operation reduces GPU requirements by avoiding an additional allocation of a `scene_width` $\times$ `scene_height` $\times$ N matrix and complex matrix masking to insert the appropriate values. The result of this modified step is that sparsity within the point cloud representation is preserved in the pseudoimage representation output by the Pillar Feature Net, allowing for its exploitation when fed into our modified Backbone.

2.2 Replacement Backbone

The Original PointPillars Backbone takes in a dense tensor format pseudoimage and employs a convolutional backbone (Appendix B, Figure 6a) akin to a feature pyramid network [11]. This backbone emits a single large pseudoimage of half the width and height of the input pseudoimage, composed of intermediary pseudoimages from the three layers of the backbone concatenated along the channel axis. Figure 3 shows an example of these intermediary pseudoimages: Figure 3a shows the input pseudoimage, and Figures 3b–3d show the smearing of non-zero entries across zero entries in the pseudoimage as it travels through the backbone. This propagation allows the network to combine information encoded in non-contiguous input pixels in order to inform the final bounding box regression, but in the process it destroys the natural sparsity seen in the input pseudoimage.

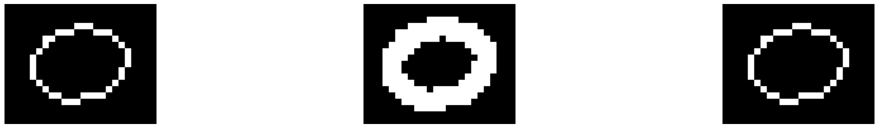


(a) Input pseudoimage (b) After `Conv` block 1 (c) After `Conv` block 2 (d) After `Conv` block 3

Figure 3: Pseudoimages from Baseline PointPillars with `BatchNorm` removed; black represents zero entries on all channels and white represents at least one non-zero channel entry. With `BatchNorm` kept in place, sparsity is entirely destroyed as zero entries are modified during normalization.

Our Sparse PointPillars Backbone takes in a sparse tensor format pseudoimage from the modified Feature Net and employs a convolutional backbone (Appendix B, Figure 6b) that uses sparse convolutions [21] and submanifold (SubM) convolutions [6] in order to preserve pseudoimage sparsity while still allowing much of the same propagation of information between non-zero entries. This is possible in large part due to SubM convolutions; the result of a sparse convolution (Figure 4b) is mathematically equivalent to the result of a standard convolution, but the sparse tensor format allows regions of all zero entries to be skipped to save computation. However, like with standard convolutions, sparse convolutions can result in smearing of non-zero entries across zero entries in the pseudoimage like Figure 3, destroying sparsity. The result of a SubM convolution (Figure 4c) is *not* mathematically equivalent to a standard convolution, allowing it to preserve sparsity, but at the

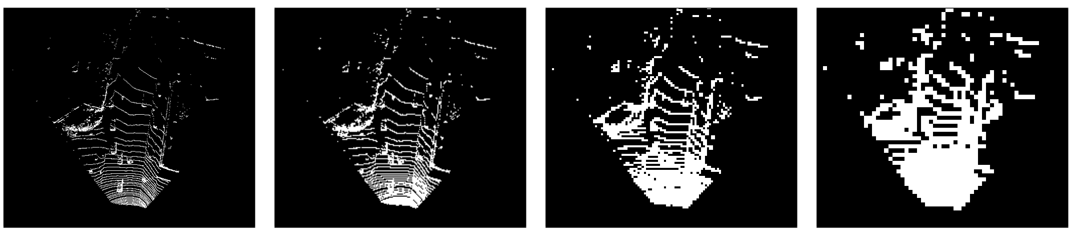
[4] `https://pytorch.org/docs/stable/generated/torch.sparse_coo_tensor.html`



(a) Input image (b) After 3×3 Standard Conv (c) After 3×3 SubM Conv

Figure 4: 3×3 stride-1 Standard Convolution versus 3×3 Submanifold (SubM) Convolution. Black represents zero entries on all channels and white represents at least one non-zero channel entry. Standard convolutions can be centered on zero entries next to non-zero entries, resulting in a new non-zero entry, causing smearing and destroying spasity. SubM convolutions are only centered on non-zero entries, preventing smearing and preserving sparsity.

cost of only allowing information to smear across existing non-zero entries. Our Backbone leverages these properties to perform similar computations to the Original PointPillars Backbone while better preserving pseudoimage sparsity (as shown in Figure 5) in order to obtain the performance benefits of using sparse operations. The combination of our modified Feature Net and Backbone allows Sparse PointPillars to maintain and exploit sparsity throughout the pipeline, without the unnecessary cost of converting sparse data into a dense representation for convolutions that will waste computation.



(a) Input pseudoimage (b) After `Conv` block 1 (c) After `Conv` block 2 (d) After `Conv` block 3

Figure 5: Pseudoimages from Sparse PointPillars. Black represents zero entries on all channels and white represents at least one non-zero channel entry. Due to the use of SubM convs and `BatchNorm` only operating over non-zero entries, sparsity is preserved across the pseudoimages.

3 Empirical Evaluation

To evaluate the performance and runtime of our Sparse PointPillars versus the original, we trained both networks on the car detection task from the KITTI [5] benchmark using the 50-50 split outlined in the PointPillars paper. Our evaluation follows the prescribed KITTI evaluation protocol of measuring the average precision (AP) at a detection threshold of 70% Intersection over Union (IoU) of the bounding box with respect to ground truth on three benchmarks: the 3D bounding box projected into image space (*BBox AP*), the bounding boxes from a BEV (*BEV AP*), and the full 3D bounding boxes (*3D AP*). Results are separated for the three KITTI difficulty levels (Easy, Medium, and Hard).

Additionally, to better understand our contributions, we perform two types of ablative studies:

1. We replaced the later sections of the sparse backbone with their dense counterparts from the original backbone to construct two variants. Using Figure 6's `Conv` block definitions, the ablated variant *Sparse1+Dense23* uses the sparse `Conv` block 1 and dense `Conv` blocks 2 and 3, and the variant *Sparse12+Dense3* uses sparse `Conv` blocks 1 and 2 with a dense `Conv` block 3.
2. We modified the filter size of the first SubM convolution of `Conv` block 2 and block 3 to be 9×9 in order to simulate the information transfer caused by pseudoimage smearing in the original model. We refer to this variant as *Sparse+WideConv*.

The Feature Extractor and Head are identical for all models (their runtimes are included in Table 1 for context), whereas all non-original models have the same Feature Net as specified in Section 2.1 and Backbones as according to their method definition. Appendix A provides further details on the experimental settings including training configuration and runtime environment details.

3.1 Runtime Analysis

The runtime for each component of each method is reported in Table 1. As shown, Sparse PointPillars is roughly 2.8 milliseconds faster than the Original PointPillars due to a roughly 0.8 millisecond faster Feature Net and a roughly 2.1 millisecond faster Backbone. Unfortunately, both ablative variants Sparse1+Dense23 and Sparse12+Dense3 are slower than the original approach, with Sparse12+Dense3 being significantly slower. We believe that these slowdowns are caused by converting the pseudoimage to a dense representation in the middle of Backbone evaluation, causing issues with pipelining of GPU operations; however, more investigation is required. Additionally, we believe that the runtime of Sparse PointPillars can be significantly further improved by using a different sparse operations library, as discussed in Appendix A.

Table 1: Model runtime in milliseconds for each network component, averaged over ten trials, run on the KITTI datset with 16cm$\times$16cm pillars. All models have the same Feature Extractor and Head (runtimes included for completeness), and all non-Original models have the same sparse Feature Net.

	Feat. Extr.	**Feat. Net**	**Backbone**	Head	Total vs Original
Original PointPillars	$6.904_{\pm 0.018}$	$1.344_{\pm 0.043}$	$16.185_{\pm 0.053}$	$3.638_{\pm 0.022}$	–
Sparse PointPillars	$6.879_{\pm 0.016}$	$\mathbf{0.508}_{\pm 0.030}$	$\mathbf{14.090}_{\pm 0.057}$	$3.778_{\pm 0.018}$	**-2.817**
Sparse1+Dense23	$6.898_{\pm 0.017}$	$\mathbf{0.517}_{\pm 0.022}$	$17.321_{\pm 0.050}$	$3.646_{\pm 0.021}$	0.223
Sparse12+Dense3	$6.973_{\pm 0.089}$	$\mathbf{0.498}_{\pm 0.021}$	$22.091_{\pm 0.245}$	$3.578_{\pm 0.063}$	5.069
Sparse+WideConv	$6.858_{\pm 0.015}$	$\mathbf{0.480}_{\pm 0.022}$	$17.483_{\pm 0.071}$	$3.684_{\pm 0.030}$	0.434

3.2 Performance Analysis

This section presents *preliminary* performance results using the training schedules and hyperparameters optimized for the Original PointPillars; we expect future hyperparameter tuning will lead to better performance. The absolute percentage of Average Precision (% AP) for the Original PointPillars on each benchmark and the relative performance of Sparse PointPillars are shown in Table 2; the relative performances of the ablative models are shown in Table 3. Even without hyperparameter tuning, Sparse PointPillars is able to perform slightly better than Original on *Easy BBox* AP and *BEV* AP, but otherwise sees drops in performance of up to 5.3% AP. Again, we anticipate that tuning the hyperparameters for the new sparse pipeline will help alleviate this decrease.

To demonstrate a lack of information propagation in Sparse PointPillars's Backbone is not the primary cause of the decreased performance, we created the Sparse+WideConv variant which features a wide SubM convolution capable of reaching the other pseudoimage pixels that could be smeared into that pixel by the Original model. Sparse+WideConv had similar performance to Sparse PointPillars, suggesting that poor hyperparameter tuning is likely the root cause of the lower performance of Sparse PointPillars on 3D AP, not a lack of information propagation.

The performance of Sparse1+Dense23 and Sparse12+Dense3 mostly lie between the results of Sparse PointPillars and Original. However, there is unusual performance degradation in some categories (e.g. Sparse1+Dense23's Easy 3D AP), further supporting the need for hyperparameter tuning.

Table 2: Performance of the original PointPillars as % AP and of our sparse model as the relative % AP difference ($\triangle$) from Original on KITTI with 16cm$\times$16cm pillars. Higher is better.

	Original PointPillars			**Sparse PointPillars**		
	Easy	Medium	Hard	Easy	Medium	Hard
BBox AP	90.51	88.67	87.06	$0.11\triangle$	$-2.68\triangle$	$-4.78\triangle$
BEV AP	89.93	87.03	84.09	$0.25\triangle$	$-5.30\triangle$	$-4.35\triangle$
3D AP	86.46	76.29	69.73	$-1.85\triangle$	$-5.31\triangle$	$-1.39\triangle$

Table 3: Ablative model % AP difference ($\triangle$) from Original on KITTI with 16cm$\times$16cm pillars.

	Sparse1+Dense23			**Sparse12+Dense3**			**Sparse+WideConv**		
	Easy	Med.	Hard	Easy	Med.	Hard	Easy	Med.	Hard
BBox AP	$-0.17\triangle$	$-0.35\triangle$	$-0.68\triangle$	$-0.23\triangle$	$-0.79\triangle$	$-0.99\triangle$	$0.00\triangle$	$-2.38\triangle$	$-4.84\triangle$
BEV AP	$-0.03\triangle$	$-0.85\triangle$	$-3.58\triangle$	$-0.24\triangle$	$-1.42\triangle$	$-2.24\triangle$	$-0.06\triangle$	$-5.56\triangle$	$-2.90\triangle$
3D AP	$-5.50\triangle$	$-1.31\triangle$	$-0.75\triangle$	$-2.13\triangle$	$-1.91\triangle$	$-1.32\triangle$	$-5.94\triangle$	$-6.38\triangle$	$-2.18\triangle$

4 Conclusion and Future Work

This work serves as a proof of concept that sparsity-aware operations can be used deep within network structures and, if designed around preserving sparsity for further exploitation, can lead to runtime improvements with the same performance or a modest decrease in performance. Our results are preliminary—there is much room for performance and runtime improvements via better backbone architectures, sparse detection heads, better model hyperparameter tuning, and faster sparse libraries.

Acknowledgments

We gratefully acknowledge the useful feedback on this work from Marcel Hussing. The research presented in this paper was partially supported by the DARPA Lifelong Learning Machines program under grant FA8750-18-2-0117, the DARPA SAIL-ON program under contract HR001120C0040, and the Army Research Office under MURI grant W911NF20-1-0080. The views and conclusions in this paper are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, ARO, or the U.S. Government.

References

[1] Alex Bewley, Pei Sun, Thomas Mensink, Drago Anguelov, and Cristian Sminchisescu. Range Conditioned Dilated Convolutions for Scale Invariant 3D Object Detection. In *Conference on Robot Learning*, 2020.

[2] Zhaowei Cai and Nuno Vasconcelos. Rethinking Differentiable Search for Mixed-Precision Neural Networks. In *Proceedings of the 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[3] R. Charles, Hao Su, Kaichun Mo, and Leonidas Guibas. PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation. In *Proceedings of the 2017 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 77–85, 2017.

[4] Xiaohan Ding, Xiangyu Zhang, Ningning Ma, Jungong Han, Guiguang Ding, and Jian Sun. RepVGG: Making VGG-style ConvNets Great Again. *arXiv preprint arXiv:2101.03697*, 2021.

[5] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite. In *Proceedings of the 2012 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[6] Benjamin Graham and Laurens van der Maaten. Submanifold Sparse Convolutional Networks. *arXiv preprint arXiv:1706.01307*, 2017.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *Proceedings of the 2016 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[8] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *Proceedings of the 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2704–2713, 2018.

[9] Minje Kim and Paris Smaragdis. Bitwise neural networks. In *ICML Workshop on Resource-Efficient Machine Learning*, 2016.

[10] Alex Lang, Sourabh Vora, Holger Caesar, Lubing Zhou, Jiong Yang, and Oscar Beijbom. PointPillars: Fast Encoders for Object Detection From Point Clouds. In *Proceedings of the 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 12689–12697, 2019.

[11] Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. Feature Pyramid Networks for Object Detection. In *Proceedings of the 2017 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[12] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: Single Shot MultiBox Detector. In *Proceedings of the European Conference on Computer Vision (ECCV) 2016*, pages 21–37, 2016.

[13] Hieu Pham, Zihang Dai, Qizhe Xie, Minh-Thang Luong, and Quoc V. Le. Meta Pseudo Labels. In *Proceedings of the 2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021.

[14] Sungho Shin, Yoonho Boo, and Wonyong Sung. Fixed-point optimization of deep neural networks with adaptive step size retraining. In *Proceedings of the 2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1203–1207, 2017.

[15] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.

[16] Joanna Stanisz, Konrad Lis, Tomasz Kryjak, and Marek Gorgon. Optimisation of the PointPillars network for 3D object detection in point clouds. In *2020 Signal Processing: Algorithms, Architectures, Arrangements, and Applications (SPA)*, pages 122–127, 2020.

[17] Hugo Touvron, Andrea Vedaldi, Matthijs Douze, and Hervé Jégou. Fixing the train-test resolution discrepancy. *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[18] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. HAQ: Hardware-Aware Automated Quantization With Mixed Precision. In *Proceedings of the 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 8604–8612, 2019.

[19] M. Wise, Michael Ferguson, Daniel King, Eric Diehr, and David Dymesich. Fetch & Freight : Standard Platforms for Service Robot Applications. In *Proceedings of the 2016 International Joint Conference on Artificial Intelligence Workshop on Autonomous Mobile Service Robots*, 2016.

[20] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. Quantized Convolutional Neural Networks for Mobile Devices. In *Proceedings of the 2016 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[21] Yan Yan, Yuxing Mao, and B. Li. SECOND: Sparsely Embedded Convolutional Detection. *Sensors*, 18, 2018.

[22] Kang Zhao, S. Huang, Pan Pan, Yinghan Li, Yingya Zhang, Zhenyu Gu, and Yinghui Xu. Distribution Adaptive INT8 Quantization for Training CNNs. In *Proceedings of the Thirty-Fifth AAAI Conference on Artificial Intelligence*, 2021.

[23] Yin Zhou and Oncel Tuzel. VoxelNet: End-to-End Learning for Point Cloud Based 3D Object Detection. In *Proceedings of the 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4490–4499, 2018.

A Runtime Details

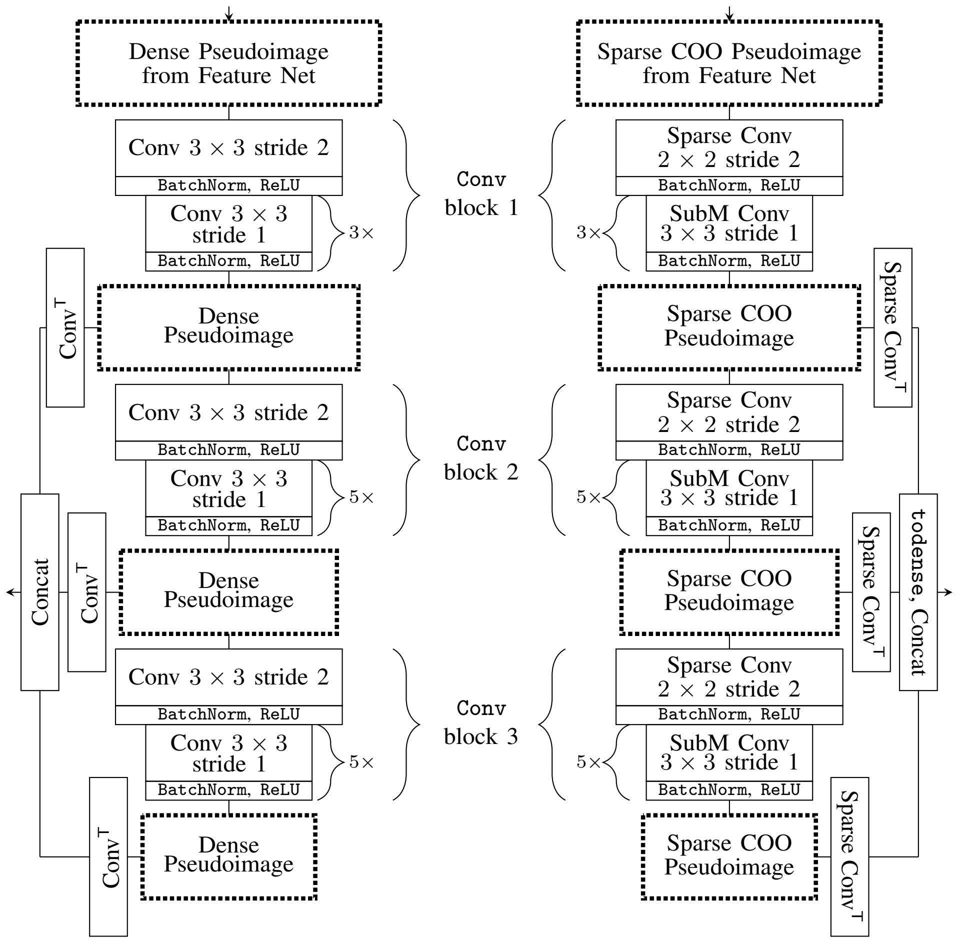
All models use 16cm$\times$16cm pillars, the standard configuration for PointPillars, and are trained using the training hyperparameters and schedule from the official PointPillars implementation. It is likely that hyperparameter tuning specifically for the Sparse PointPillars model and the ablative models will result in better performance, but they will not impact evaluation runtimes.

For the sparse convolution and SubM convolution implementation, we use `spconv`[5], the sparse convolution library provided by the authors of SECOND [21], the codebase upon which the official implementation of PointPillars is built. Due to the implementation of `spconv`, sparse convolutions and SubM convolutions have significant overhead, and we believe switching to a more optimized library in the future will significantly improve our method's runtime.

[5] `https://github.com/traveller59/spconv`

All evaluation runtimes are measured on a dedicated Ubuntu system with an NVidia 2080ti GPU and an AMD Ryzen 7 3700X CPU using CUDA 10.1 with the PyTorch 1.5 runtime and averaged over ten runs. The official runtimes reported in the PointPillars paper and the KITTI leaderboards use NVidia's TensorRT runtime, leading to a reported 40% performance improvement over PyTorch's runtime, but using TensorRT takes substantial engineering effort when using custom layers from libraries like `spconv`.

B Full Backbone Architectures



(a) Original Backbone (b) Sparse PointPillars Backbone

Figure 6: Original vs Sparse PointPillars PointPillars Backbone. The Sparse PointPillars Backbone is a modified version of Original Backbone designed to preserve and exploit pseudoimage sparsity.