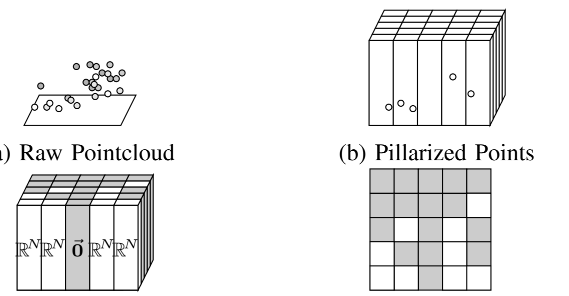
Sparse PointPillars: Maintaining and Exploiting Input Sparsity to Improve Runtime on Embedded Systems

Kyle Vedder* and Eric Eaton†

***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 maintain *and* exploit this input sparsity, leading to decreased runtimes. We present results on KITTI, a canonical 3D detection dataset, and Matterport-Chair, a novel Matterport3D-derived chair detection dataset from scenes in real furnished homes. We evaluate runtime characteristics using a desktop GPU, an embedded ML accelerator, and a robot CPU, demonstrating that our method results in significant runtime decreases (2× or more) for embedded systems with only a modest decrease in detection quality. Our work represents a new approach for practitioners to optimize models for embedded systems by maintaining *and* exploiting input sparsity throughout their entire pipeline to reduce runtime and resource usage while preserving detection performance. All models, weights, experimental configurations, and datasets used are publicly available[1].**



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

Fig. 1: PointPillars' pillarization and Bird's Eye View (BEV) pseudoimage creation process that naturally captures input sparsity. PointPillars processes the Sparse BEV Pseudoimage as a dense tensor, while Sparse PointPillars maintains and exploits the input sparsity by processing it as a sparse tensor, leading to reduced resource usage and increased speed.

I. Introduction

Robot development tends to fall into two stages: first, resource utilization and power limits are ignored in order to develop a functional system and then, second, these systems are scaled down to fit within commercially viable limits of power, memory, and computation without significantly sacrificing system performance. Many autonomous vehicle manufacturers are currently facing this second challenge and it is even more pronounced for developers of intelligent mobile robots, where resources are limited from the very start. For example, even a larger, high-end robot like the Fetch Freight [1] is not able to support several desktop-grade GPUs plus high-end CPUs in order to run its control stack, forcing roboticists to settle for smaller embedded systems like NVidia's Jetson [2], while 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.

In this paper, we address this problem of reducing resource usage while preserving performance within PointPillars [3], a popular 3D object detector that operates on point clouds from a Bird's Eye View (BEV). PointPillars segregates the raw points (Fig. 1a) into pillars (Fig. 1b) and vectorizes these point collections into N dimensional vectors with empty pillars represented by the zero vector $\vec{\mathbf{0}} = \{0\}^N$ (Fig. 1c), resulting in a sparse BEV pseudoimage of the scene (Fig. 1d). PointPillars processes the pseudoimage with a dense 2D convolutional backbone (Fig. 6a) and then predicts bounding boxes using Single-Stage Detector (SSD) [4]. The PointPillars Backbone is the most expensive component of the pipeline, but the pseudoimage it processes is highly sparse. As a result, large, empty sections of the image are unnecessarily 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 on embedded systems while maintaining reasonable performance. Concretely, we propose two changes (Fig. 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 III-A), and
2) A new Backbone that maintains *and* exploits the natural sparsity of the pseudoimage via sparse [5] and submanifold [6] convolutions (Section III-B).

The rest of this work proceeds as follows: we discuss relevant approaches to speeding up inference (Section II) and present our pipeline modifications (Section III), theoretically analyze their computational savings (Section IV) and empirically evaluate these savings across multiple datasets

*Kyle Vedder is with the Dept. of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, USA kvedder@seas.upenn.edu

†Eric Eaton is with the Dept. of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, USA eeaton@seas.upenn.edu

[1] https://vedder.io/sparse_point_pillars

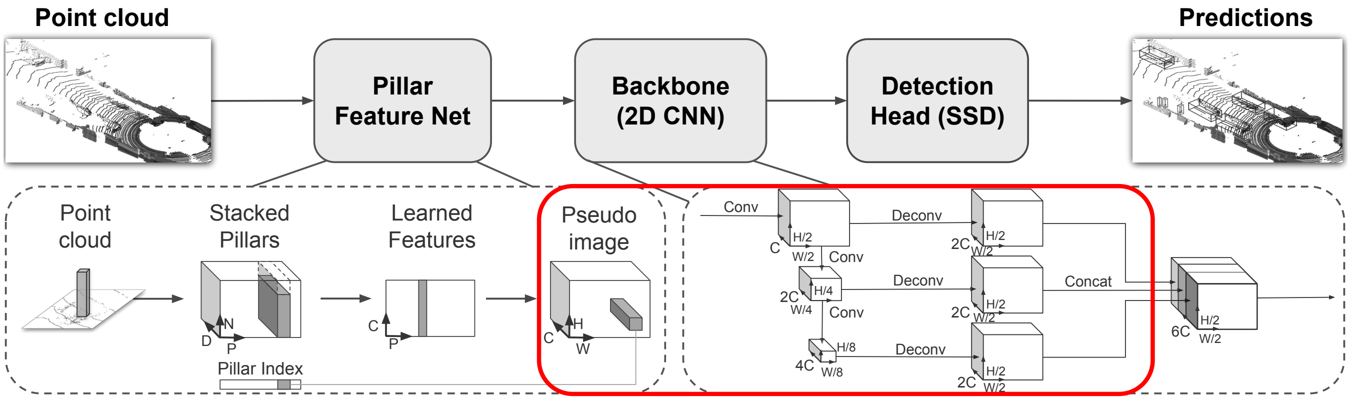


Fig. 2: PointPillars [3] pipeline with our Sparse PointPillars's modified sections circled in red. Figure adapted from [3].

and compute platforms. Our results demonstrate that our pipeline changes provide a 2× or more speedup on embedded systems in exchange for a modest decrease in detection quality (Section V). We then conclude with future work (Section VI).

II. Related Work

Solutions to the problem of reducing resource usage in order to deploy machine learning models on embedded hardware typically fall into three categories. One general-purpose solution to this problem, model quantization [7]–[11], first trains models in a standard fashion using floating point weights and then, after training, converts some [7] or all [8], [9] weights into integer [10] or binary [11] quantized values that are faster to multiply than floating point values. The quantized network is then finetuned, resulting in similar performance while running faster on accelerators (e.g., GPUs [12]), low-end compute devices (e.g., mobile phones [13]), or specialized hardware (e.g., FPGAs [14]).

A second model-agnostic approach, often paired with quantization, is model *weight* sparsification, also known as model pruning. As per the Lottery Ticket Hypothesis [15], in many models the majority of its weights contribute little to its performance [16]–[18] and can be pruned to improve runtime [19]–[25], either with regular structure to exploit hardware properties [20]–[22], or without structure [23]–[25].

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 [26]–[28]. 3D convolution of these voxels dominates network runtime [26], but the voxels tend to be highly sparse; point clouds from KITTI contain over 100,000 points, but when voxelized into 16cm cubes, over 95% are empty [3], [27]. This motivates sparsity-aware convolutional methods such as SECOND [27], which performs sparse 3D convolutions to produce mathematically identical results with significantly less computation and runtime. SBNet [29], a BEV 3D object detector, takes a similar approach by doing dense convolutions only in coarsely masked regions of relevance to reduce the area convolved, decreasing runtime without a significant impact on performance. This paper augments this set of sparsity exploitation tactics by developing a pipeline that maintains *and* exploits input sparsity for PointPillars [3] to obtain significant runtime improvements.

III. PointPillars Modifications to Maintain and Exploit End-to-End Sparsity

This section describes our modifications to PointPillars in order to maintain *and* exploit sparsity throughout the processing pipeline. We then theoretically analyze (Section IV) and empirically validate (Section V) our modifications.

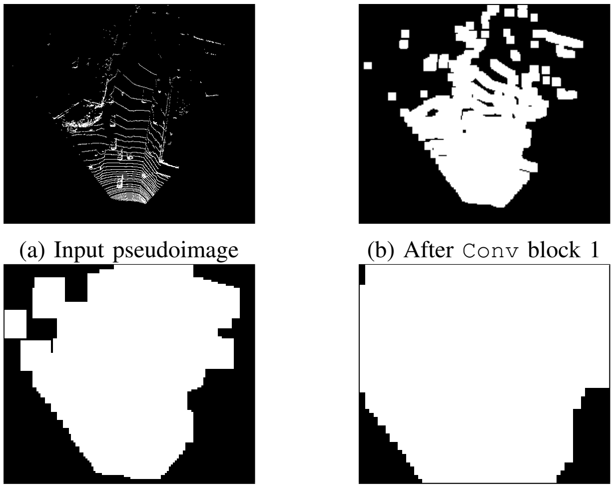
A. Replacement Pillar Feature Net

PointPillars' pillerization process (Fig. 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 [30]. 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 [31] using the coordinate information already recorded during the *gather* step. This constant-time operation reduces GPU requirements by avoiding an additional allocation of a $\mathrm{scene\_width} \times \mathrm{scene\_height} \times N$ matrix plus complex matrix masking to insert the appropriate values *and* allows the Pillar Feature Net to emit a sparse pseudoimage output, enabling its further exploitation by our modified Backbone.

B. Replacement Backbone

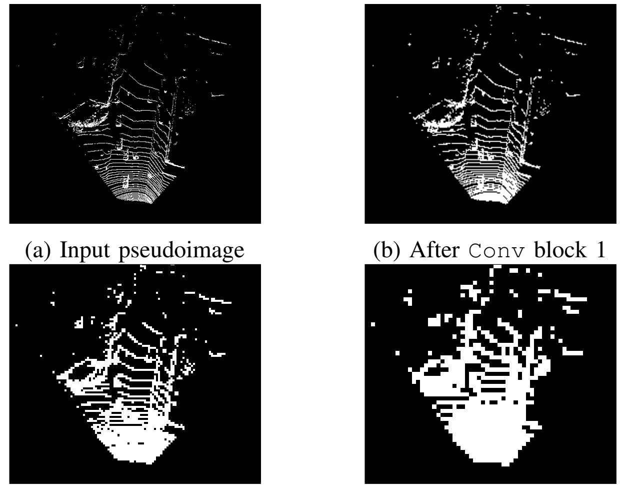
The original PointPillars Backbone takes in the dense tensor format pseudoimage from the *scatter* step and processes it with a convolutional feature pyramid network [32] style backbone (Fig. 6a). This backbone emits a single large pseudoimage of half the width and height of the input pseudoimage, composed of intermediary pseudoimages from the three layers of the backbone concatenated along the channel axis. Due to the heavy use of standard 3×3 stride-1 convolutions throughout the Backbone, there is significant smearing of non-zero entries across the pseudoimage; Fig. 3



(a) Input pseudoimage (b) After `Conv` block 1

(c) After `Conv` block 2 (d) After `Conv` block 3

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



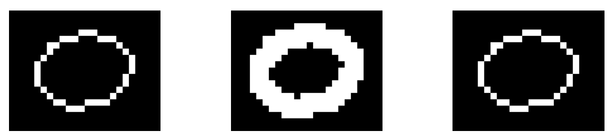
(a) Input pseudoimage (b) After `Conv` block 1

(c) After `Conv` block 2 (d) After `Conv` block 3

Fig. 4: Pseudoimages from our Sparse PointPillars run on a sample from KITTI. 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 maintained.

visualizes this for an input from the KITTI dataset: Fig. 3a shows the input pseudoimage, and Figs. 3b–3d show the smearing of non-zero entries across zero entries in the pseudoimage as it travels through the backbone.

Our Sparse PointPillars' Backbone (Fig. 6b) is a modified version of the original PointPillars' Backbone, but takes in the COO sparse tensor pseudoimage from our modified Pillar Feature Net and uses sparse convolutions [27] and submanifold (SubM) convolutions [6] to maintain and exploit the pseudoimage's sparsity. Fig. 4 visually demonstrates maintenance of the pseudoimage's sparsity as it flows through the Sparse PointPillars Backbone. Compared to PointPillars' Backbone, Sparse PointPillars' Backbone maintains and exploits sparsity in two key ways: 1) it replaces the 3×3 stride-2 convolutions with 2×2 stride-2 convolutions when shrinking the pseudoimage, ensuring that non-zero entries in the higher resolution pseudoimage appear only once in the lower resolution pseudoimage, and 2) it replaces the 3×3 stride-1 convolutions with 3×3 stride-1 SubM convolutions. The result of a sparse convolution (Fig. 5b) is mathematically equal to the result of a standard convolution but with zero entries skipped to save computation, causing the same smearing of non-zero entries across zero entries in the pseudoimage as dense convolutions (e.g., Fig. 3), destroying sparsity. The result of a SubM convolution (Fig. 5c) is *not* mathematically equal to a standard convolution; SubM convolutions only convolve with filters centered on existing non-zero entries, allowing the backbone to *maintain* the level of sparsity while processing the pseudoimage in its SubM blocks (the output of a stride-1 SubM convolution will have the exact same sparsity as the input) and to *exploit* this sparsity to perform significantly fewer computations.

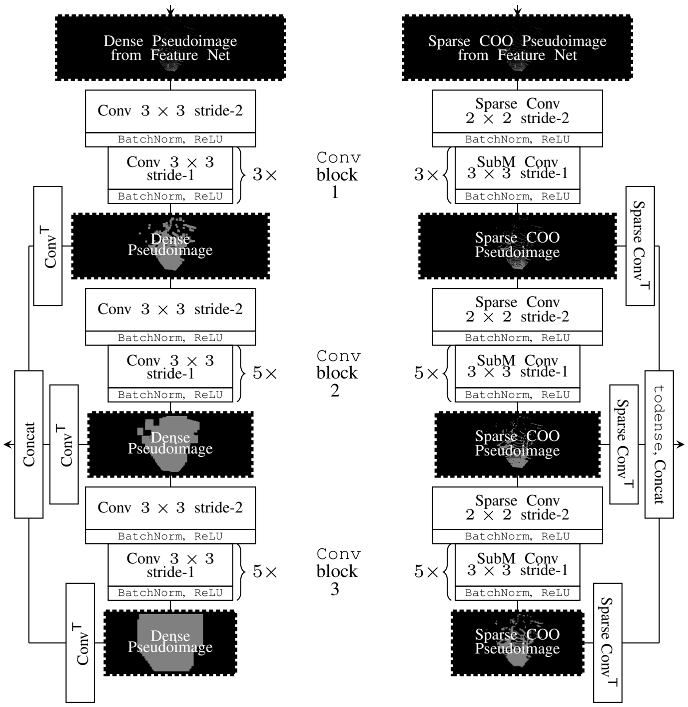


(a) Input image (b) After 3×3 Standard Conv (c) After 3×3 SubM Conv

Fig. 5: 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 maintaining sparsity.

IV. Theoretical Analysis

Our replacement Backbone maintains and exploits input sparsity to perform fewer operations in order to achieve faster runtime compared to PointPillars' Backbone. For PointPillars, the number of convolutions of its Backbone (Fig. 6a) is a function of the area and number of channels of the input pseudoimage; the *values* of the input pseudoimage are irrelevant. By comparison, the number of convolutions of Sparse PointPillars' Backbone (Fig. 6b) is a function of the area, number of channels, *and* the pseudoimage density (the fraction of non-zero values of the input pseudoimage). Due to the strategic use of 2×2 stride-2 convolutions in halving the width and height of the pseudoimage, Sparse PointPillars' Backbone increases the pseudoimage density by *at most* $4\times$ per halving of pseudoimage size; however, in practice this density increase is far below $4\times$ as non-zero entries tend to appear non-uniformly next to one another (e.g., Fig. 4).



(a) PointPillars (b) Sparse PointPillars

Fig. 6: PointPillars vs Sparse PointPillars Backbone. The Sparse PointPillars Backbone maintains and exploits pseudoimage sparsity by using SubM convs and 2×2 stride-2 convs to avoid the smearing effect of 3×3 stride-1 convs.

Table I outlines the type and number of convolutions performed by each Backbone, providing an exact count for PointPillars and an *upper bound* for Sparse PointPillars. When the input pseudoimage density D approaches 1, Sparse PointPillars' Backbone collapses to PointPillars' backbone with the 3×3 stride-2 convolutions replaced with 2×2 stride-2 convolutions; when D approaches 0, Sparse PointPillars' Backbone performs significantly fewer convolutions.

In practice, input density D is very small; for KITTI's test data, the median D is 0.02459 (min 0.013321, max 0.03899) and for Matterport-Chair's test data, the median D is 0.00750 (min 0.00029, max 0.01679). Translating this into an convolution operations count, for KITTI's median density, Sparse PointPillars' Backbone performs *at least* 50% fewer convolutions, and for Matterport-Chair's median density, Sparse PointPillars' Backbone performs *at least* 79% fewer convolutions. In general, the larger the max range on the depth sensor or the smaller the pillar size, the faster Sparse PointPillars will run compared to PointPillars; pseudoimage area scales quadratically with both the maximum sensor range and pillar size, and points at range tend to be farther apart, leading to more ragged detections that can be exploited by Sparse PointPillars' Backbone.

V. Empirical Evaluation

To validate the design of our modified Backbone, we implemented it in `Open3D-ML` [33], a high-quality third party implementation of PointPillars, using the Minkowski Engine [5] for sparse convolutions. We evaluate Sparse PointPillars against PointPillars on the KITTI [34] dataset, a well understood 3D self-driving dataset used for evaluation in the original PointPillars paper (Section V-A). We perform several ablative studies on our Backbone to demonstrate that it produces a reasonable trade-off between runtime and detection performance. To demonstrate Sparse PointPillars' value on embedded systems via a realistic task, we then compare it against PointPillars on *Matterport-Chair*, a custom chair detection task derived from Matterport3D [35], an indoor 3D scan dataset designed to simulate a task required of real service robots (Section V-B). We evaluate the two Matterport-Chair models across three compute platforms: a desktop with a high-end GPU, an embedded ML accelerator configured for minimal and maximal power modes, and the CPU of a high-end commercial robot.

TABLE I: Number of convolutions performed by PointPillars' Backbones and an *upper bound* on number of convolutions performed by Sparse PointPillars' Backbone, for an input pseudoimage of size $W \times H$ with C channels and D density.

Operation	Baseline Count	Sparse PointPillars's Upper Bound
3×3 Conv	$\frac{15}{4}C^2HW$	$C^2HW(\min(\frac{3}{4}, 3D) + \min(\frac{5}{4}, 20D) + \min(\frac{5}{4}, 80D))$
2×2 Conv	0	$C^2HW(\frac{D}{4} + \min(\frac{1}{8}, \frac{1}{2}D) + \min(\frac{1}{8}, 2D))$
1×1 Conv^T	$\frac{1}{2}C^2HW$	$C^2HW\min(\frac{1}{2}, 2D)$
2×2 Conv^T	$\frac{1}{4}C^2HW$	$C^2HW\min(\frac{1}{4}, 4D)$
4×4 Conv^T	$\frac{1}{8}C^2HW$	$C^2HW\min(\frac{1}{8}, 8D)$

A. KITTI Evaluation with Ablative Studies

KITTI [34], a self-driving car dataset of LiDAR point clouds with human-annotated 3D bounding boxes, is a common benchmark dataset in 3D object detection and is used as the evaluation dataset in the original PointPillars paper. Both Sparse PointPillars and PointPillars are trained on the KITTI Car detection task, configured with the default 16cm×16cm pillars, 504×440 pillar pseudoimage, 50%/50% train/validation split, and hyperparameter configurations outlined in the PointPillars paper, with the exception that Sparse PointPillars performs 50 more epochs in order to converge. Despite performing 25% more epochs, Sparse PointPillars trains in roughly the same amount of time. 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 relative to ground truth on two key benchmarks: the bounding boxes from a BEV (*BEV AP*) and the full 3D bounding boxes (*3D AP*). KITTI does not have public labels for its test set, so in keeping with the literature [3], [26], [27] we report results on the validation set. Results are separated for the three KITTI difficulty levels (Easy, Medium, Hard), and runtimes are recorded on a dedicated desktop with an AMD Ryzen 7 3700X CPU and an NVidia 2080ti GPU.

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

1) We replace the later sections of the sparse Backbone with their dense counterparts from the original Backbone to construct two variants. Using Fig. 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 modify the filter size of the first SubM convolution of each `Conv` block 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 absolute percentage of Average Precision (% AP) for PointPillars on each benchmark and the relative performance of Sparse PointPillars and its ablations are shown in Table III. Relative to PointPillars, Sparse PointPillars performs roughly 5% AP worse on *BEV* and roughly 8.5% AP worse on *3D*, and roughly equally to the ablative models, with Sparse1+Dense23 performing slightly better and with Sparse12+Dense3 and Sparse+WideConv perform worse. Together, these results indicate that SubM convolutional blocks in the Backbone are more difficult to train, even if the block has access to the same information as the dense model.

The runtime for each component of each method is reported in Table II. Sparse PointPillars is 0.18ms faster than PointPillars. Our Feature Net runs 0.18ms faster as it avoids the *scatter* step, but the recorded runtime for our Backbone is actually 5.22ms *slower* than PointPillars Backbone; however, the BBox Extract stage, despite running *identical* code, is 5.22ms *faster* in Sparse PointPillars than in PointPillars. The runtime difference comes from the time taken to allocate the memory for the anchor boxes—both allocate the same size GPU array, but due to pipelining and earlier memory cleanup that inflated the Sparse PointPillars' Backbone's runtime, it is able to allocate the final anchor boxes faster. This phenomenon recurs with Matterport-Chair on other GPU accelerated devices tested in Section V-B. Unsurprisingly, Sparse1+Dense23 and Sparse12+Dense3 are both slower than PointPillars and Sparse PointPillars due to the Backbone pipelining interruption when converting from a sparse to a dense tensor, and Sparse+WideConv is significantly slower due to its very large convolutions.

B. Matterport-Chair Evaluation with Embedded Performance

To simulate a realistic detection task faced by a service robot or other embodied platform using embedded compute systems, we constructed a labeled chair detection dataset *Matterport-Chair* using point clouds and their object labels sampled from houses in Matterport3D [35]. Matterport3D is a dataset of multiple building-scale indoor 3D meshes constructed using many high-resolution panoramic RGBD views taken inside real houses and labeled with 3D bounding boxes and semantic labels for over 20 different object classes. To generate our training and test dataset, we sampled point clouds of random views from the perspective of a robot sitting one meter off the ground across four different high quality house meshes, producing a train/test split of 7,500 point clouds each (the same size as the KITTI splits). We post-processed the bounding boxes, aligning them vertically and rejecting boxes that were highly occluded, associated with too few points, or caused by dataset noise such as holes in the house mesh. The training and test splits along with the generation code are available on the project webpage.

Both PointPillars and Sparse PointPillars are trained with 5cm$\times$5cm pillars and 768×512 pillar pseudoimage (set using the max absolute X and Y-axis values of the training data point clouds), 66%/33% train/validation splits, and standard hyperparameters, with the exception that Sparse PointPillars again performs 50 more epochs in order to converge. Despite performing 25% more epochs, Sparse PointPillars trains $1.8\times$ faster. Our evaluation follows the KITTI protocol of measuring the average precision (AP) at a detection threshold of 50% Intersection over Union (IoU) of the bounding box relative to ground truth on two key benchmarks: the bounding boxes from BEV (*BEV AP*) and in full 3D (*3D AP*).

Sparse PointPillars lags behind PointPillars in performance by 6.04% AP on *BEV* and by 4.61% AP on *3D*.[2] However, as shown in Table IV, due to Matterport-Chair's low density, Sparse PointPillars is significantly faster than PointPillars across the full range of compute platforms available to embodied agents: a desktop with an AMD Ryzen 7 3700X CPU and an NVidia 2080ti GPU (the same system used to benchmark KITTI, denoted *Desktop*), an NVidia Jetson Xavier embedded ML accelerator configured for the highest and lowest power settings (30 Watt, 8 core mode denoted *Xavier High* and 10 Watt, 2 core mode denoted *Xavier Low*), and a Fetch Freight [1] robot's built-in four core Intel i5-4590S CPU (denoted *Robot*). Sparse PointPillars is more than $1.5\times$ as fast as PointPillars on *Desktop* (and fast enough for 60Hz inference), more than $2\times$ as fast on *Xavier High* (and fast enough for 10Hz inference), almost $3\times$ as fast on *Xavier Low* (and fast enough for 6Hz inference), and more than $4\times$ as fast on *Robot* (and fast enough for 4Hz inference).

Like on KITTI, the Backbone runtimes for Sparse PointPillars on GPU accelerated platforms is *slower* than PointPillars, but the BBox Extract stage is *faster* despite using *identical* code due to differences in memory allocation speed in different parts of the runtime pipeline; however, the Robot evaluations demonstrate that when GPU pipelining is not a factor, Sparse PointPillars' Backbone is far faster than PointPillars' Backbone, and the BBox Extract stage runs at roughly the same speed.

VI. Conclusion and Future Work

This work demonstrates that Sparse PointPillars allows practitioners to trade small amounts of model performance for significant decreases in runtime and resource usage on embedded systems. For example, on our Matterport-Chair dataset, Sparse PointPillars runs faster on the Jetson Xavier in low power mode than PointPillars does in high power mode, allowing a practitioner to save power *and* get reduced runtimes at the cost of a few % AP. Alternatively, PointPillars runs at

[2] PointPillars achieved 84.09% AP on *BEV* and 80.66% AP on *3D*

TABLE II: Per instance model component runtime and standard deviation in milliseconds, run on KITTI's test set, averaged over ten trials. Models differ only in their Feature Net and Backbone; all other components are identical. Lower is better.

	To Device	Feature Extract	**Feature Net**	**Backbone**	Head	BBox Extract	**Total time**
PointPillars	0.064 ± 0.000	2.375 ± 0.010	0.303 ± 0.004	2.358 ± 0.027	0.204 ± 0.002	9.105 ± 0.030	14.410 ± 0.033
Sparse PointPillars	0.064 ± 0.000	2.330 ± 0.009	0.133 ± 0.002	7.578 ± 0.038	0.244 ± 0.004	3.877 ± 0.010	**14.226 ± 0.056**
Sparse1+Dense23	0.064 ± 0.000	2.341 ± 0.011	0.134 ± 0.002	7.394 ± 0.049	0.231 ± 0.003	4.437 ± 0.013	14.602 ± 0.068
Sparse12+Dense3	0.064 ± 0.000	2.359 ± 0.011	0.134 ± 0.002	7.803 ± 0.050	0.236 ± 0.002	4.406 ± 0.018	15.001 ± 0.080
Sparse+WideConv	0.066 ± 0.001	2.356 ± 0.015	0.137 ± 0.002	17.286 ± 0.071	0.242 ± 0.007	6.184 ± 0.038	26.270 ± 0.124

TABLE III: Performance of PointPillars as % AP and performance of Sparse PointPillars and its ablations as the relative % AP difference (△) to PointPillars on KITTI with 16cm×16cm pillars. Higher is better.

	PointPillars			**Sparse PointPillars**			**Sparse1+Dense23**			**Sparse12+Dense3**			**Sparse+WideConv**		
	Easy	Med.	Hard	Easy	Med.	Hard	Easy	Med.	Hard	Easy	Med.	Hard	Easy	Med.	Hard
BEV AP	90.75	89.79	89.48	-5.71△	-6.53△	-4.98△	-5.25△	-5.44△	-3.73△	-5.54△	-5.89△	-5.60△	-6.89△	-7.24△	-9.25△
3D AP	82.30	80.34	79.11	-8.82△	-7.83△	-9.16△	-7.62△	-7.36△	-8.85△	-8.88△	-7.71△	-9.27△	-11.1△	-11.57△	-13.69△

TABLE IV: Per instance model component runtime and standard deviation in milliseconds, run on Matterport-Chair's test set, averaged over ten trials for Desktop and Xavier High and averaged over three trials for Xavier Low and Robot. Models differ only in their Feature Net and Backbone; all other components are identical. Lower is better.

	To Device	Feature Extract	**Feature Net**	**Backbone**	Head	BBox Extract	**Total time**
Desktop Dense	0.072 ± 0.001	1.783 ± 0.010	0.390 ± 0.005	2.512 ± 0.017	0.199 ± 0.001	15.198 ± 0.145	20.154 ± 0.143
Desktop Sparse	0.073 ± 0.001	1.747 ± 0.015	0.130 ± 0.002	6.633 ± 0.038	0.218 ± 0.002	4.854 ± 0.010	**13.655 ± 0.060**
Xavier High Dense	0.657 ± 0.056	13.526 ± 0.195	4.509 ± 0.074	17.335 ± 0.163	1.586 ± 0.016	126.374 ± 0.227	163.987 ± 0.485
Xavier High Sparse	0.602 ± 0.007	13.451 ± 0.076	1.341 ± 0.030	43.935 ± 0.176	2.199 ± 0.012	27.054 ± 0.139	**88.584 ± 0.349**
Xavier Low Dense	2.128 ± 0.422	28.340 ± 0.098	6.907 ± 0.094	14.557 ± 0.292	1.511 ± 0.009	407.499 ± 0.302	460.941 ± 0.246
Xavier Low Sparse	2.163 ± 0.106	28.813 ± 0.047	1.728 ± 0.007	60.233 ± 0.076	2.385 ± 0.004	62.169 ± 0.073	**157.492 ± 0.199**
Robot Dense	1.531 ± 0.216	43.073 ± 0.482	29.237 ± 0.319	879.225 ± 6.116	115.363 ± 1.065	13.706 ± 0.064	1,082.135 ± 6.692
Robot Sparse	1.383 ± 0.111	41.073 ± 0.994	0.313 ± 0.004	66.045 ± 0.409	114.911 ± 1.171	13.012 ± 0.171	**236.737 ± 2.491**

less than 1Hz on the robot's CPU; with Sparse PointPillars, practitioners can reliably run at 1Hz *and* have more than 75% of the CPU budget left to run other components of the robot control stack. By providing faster runtimes at an architecture level, Sparse PointPillars provides practitioners new tools in their toolbox to build and optimize their full control stack.

This work can be extend by exploring model quantization and weight pruning in tandem with our Backbone. Prior art has shown significant quantization of PointPillars results in only minor drops in performance [14]. When combined with Sparse PointPillars, this may result in significant further reductions in runtime for a modest drop in performance, or enable inference on more exotic hardware (e.g., FPGAs).

Additionally, this work would benefit from further performance evaluation using a Streaming AP [36] style measure extended to 3D detectors. In this work, we evaluated detection quality with AP, a standard metric in the vision literature that matches output detections to the *input* point cloud. However, this evaluation protocol does not represent the problem practitioners face: in dynamic environments, the detection is most useful if it matches the state of the world *at the time it is emitted*. The world changes while the detector is performing inference and so a quick, lower quality detection is potentially better representative of the world upon emission than a slow, higher quality detection. A Streaming AP style measure would directly consider the dramatic latency reductions of Sparse PointPillars in the evaluation of its accuracy, better reflecting the problem formulation that practitioners face.

Acknowledgments

We gratefully acknowledge the useful feedback on this work from Marcel Hussing. This research 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] M. Wise, M. Ferguson, D. King, E. Diehr, and D. 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.

[2] D. Franklin, V. Nguyen, and R. Love, "Jetson AGX Xavier and the New Era of Autonomous Machines," 2019.

[3] A. Lang, S. Vora, H. Caesar, L. Zhou, J. Yang, and O. 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)*, 2019, pp. 12 689–12 697.

[4] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "SSD: Single Shot MultiBox Detector," in *Proceedings of the European Conference on Computer Vision (ECCV) 2016*, 2016, pp. 21–37.

[5] C. Choy, J. Gwak, and S. Savarese, "4D Spatio-Temporal ConvNets: Minkowski Convolutional Neural Networks," in *Proceedings of the 2019 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

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

[7] Z. Cai and N. 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.
[8] S. Shin, Y. Boo, and W. 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)*, 2017, pp. 1203–1207.
[9] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, “HAQ: Hardware-Aware Automated Quantization With Mixed Precision,” in *Proceedings of the 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.
[10] K. Zhao, S. Huang, P. Pan, Y. Li, Y. Zhang, Z. Gu, and Y. Xu, “Distribution Adaptive INT8 Quantization for Training CNNs,” in *Proceedings of the Thirty-Fifth AAAI Conference on Artificial Intelligence (AAAI)*, 2021.
[11] M. Kim and P. Smaragdis, “Bitwise neural networks,” in *ICML Workshop on Resource-Efficient Machine Learning*, 2016.
[12] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. 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)*, 2018.
[13] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. 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.
[14] J. Stanisz, K. Lis, T. Kryjak, and M. Gorgon, “Optimisation of the PointPillars network for 3D object detection in point clouds,” in *2020 Signal Processing: Algorithms, Architectures, Arrangements, and Applications (SPA)*, 2020, pp. 122–127.
[15] J. Frankle and M. Carbin, “The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks.” in *Proceedings of the Conference on Learning Representations, (ICLR)*, 2019.
[16] Y. Lecun, J. Denker, and S. Solla, “Optimal Brain Damage,” in *Advances in Neural Information Processing Systems (NeurIPS)*, 1989.
[17] B. Hassibi, D. Stork, and G. Wolff, “Optimal Brain Surgeon and general network pruning,” in *Proceedings of the IEEE International Conference on Neural Networks*, 1993.
[18] R. Reed, “Pruning Algorithms - A Survey,” *IEEE Transactions on Neural Networks*, vol. 4, no. 5, 1993.
[19] D. Blalock, J. J. Gonzalez Ortiz, J. Frankle, and J. Guttag, “What is the State of Neural Network Pruning?” in *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze, Eds., 2020.
[20] J. Pool, A. Sawarkar, and J. Rodge, “Accelerating Inference with Sparsity Using the NVIDIA Ampere Architecture and NVIDIA TensorRT,” 2021.
[21] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, “Learning Structured Sparsity in Deep Neural Networks,” 2016.
[22] Y. He, X. Zhang, and J. Sun, “Channel Pruning for Accelerating Very Deep Neural Networks,” in *Proceedings of the 2017 IEEE/CVF International Conference on Computer Vision (ICCV)*, 10 2017.
[23] K. Ullrich, E. Meeds, and M. Welling, “Soft Weight-Sharing for Neural Network Compression,” in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2017.
[24] J. Frankle, G. Dziugaite, D. M. Roy, and M. Carbin, “Linear Mode Connectivity and the Lottery Ticket Hypothesis,” in *Proceedings of the International Conference on Machine Learning (ICML)*, 2020.
[25] S. Han, J. Pool, J. Tran, and W. J. Dally, “Learning Both Weights and Connections for Efficient Neural Networks,” in *Advances in Neural Information Processing Systems (NeurIPS)*, 2015.
[26] Y. Zhou and O. 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*, 2018.
[27] Y. Yan, Y. Mao, and B. Li, “SECOND: Sparsely Embedded Convolutional Detection,” *Sensors*, vol. 18, 2018.
[28] A. Bewley, P. Sun, T. Mensink, D. Anguelov, and C. Sminchisescu, “Range Conditioned Dilated Convolutions for Scale Invariant 3D Object Detection,” in *Conference on Robot Learning*, 2020.
[29] M. Ren, A. Pokrovsky, B. Yang, and R. Urtasun, “SBNet: Sparse Blocks Network for Fast Inference,” in *Proceedings of the 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.
[30] R. Charles, H. Su, K. Mo, and L. 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)*, 2017.
[31] P. A. Tew, “An Investigation of Sparse Tensor Formats for Tensor Libraries,” Master’s thesis, Massachusetts Institute of Technology, Cambridge, MA, September 2016.
[32] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie, “Feature Pyramid Networks for Object Detection,” in *Proceedings of the 2017 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.
[33] Q.-Y. Zhou, J. Park, and V. Koltun, “Open3D: A modern library for 3D data processing,” *arXiv:1801.09847*, 2018.
[34] A. Geiger, P. Lenz, and R. 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.
[35] I. Armeni, O. Sener, A. R. Zamir, H. Jiang, I. Brilakis, M. Fischer, and S. Savarese, “3D Semantic Parsing of Large-Scale Indoor Spaces,” in *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.
[36] M. Li, Y. Wang, and D. Ramanan, “Towards Streaming Perception,” in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2020.