TorchSparse++: Efficient Point Cloud Engine

Haotian Tang[1,*] Shang Yang[1,2,*] Zhijian Liu[1,*]
Ke Hong[2] Zhongming Yu[4] Xiuyu Li[5] Guohao Dai[3] Yu Wang[2] Song Han[1]
[1]MIT [2]Tsinghua University [3]Shanghai Jiao Tong University [4]UCSD [5]UC Berkeley

https://torchsparse.mit.edu

Abstract

Point cloud computation has become an increasingly more important workload for autonomous driving and other applications. Unlike dense 2D computation, point cloud convolution has ***sparse*** *and* ***irregular*** *computation patterns and thus requires dedicated inference system support with specialized high-performance kernels. While existing point cloud deep learning libraries have developed different dataflows for convolution on point clouds, they assume a single dataflow throughout the execution of the entire model. In this work, we systematically analyze and improve existing dataflows. Our resulting system, TorchSparse++, achieves* ***2.9***×*,* ***3.3***×*,* ***2.2***× *and* ***1.8***× *measured end-to-end speedup on an NVIDIA A100 GPU over the state-of-the-art MinkowskiEngine, SpConv 1.2, TorchSparse and SpConv v2 in inference respectively. Furthermore, TorchSparse++ is the only system to date that supports all necessary primitives for 3D segmentation, detection, and reconstruction workloads in autonomous driving. Code is publicly released at* https://github.com/mit-han-lab/torchsparse.

1. Introduction

3D point cloud has become increasingly accessible over the past few years thanks to the widely available 3D sensors, including LiDAR scanners and depth cameras, making it a popular data representation in many real-world scenarios such as autonomous driving. It is of great importance to optimize the inference for point cloud models as these applications usually target real-time performance.

3D point clouds are sparse, rendering regular dense convolution inapplicable. Sparse convolution [12, 16] extends the definition of regular convolution by only conducting computation for non-zero features. It is arguably the most important building block for almost all state-of-the-art 3D perception models (*e.g*. 3D semantic segmentation [10, 24, 36], 3D object detection [1, 6, 8, 15, 40, 42, 44, 46], 3D reconstruction [9], multi-sensor fusion [7, 20, 23], end-to-end navigation [22]). Despite achieving dominant performance, the irregular nature of sparse convolution makes it harder to be processed on general-purpose hardware (*e.g*. GPU) as it lacks official vendor library support. Dedicated inference engines with specialized high-performance kernels are required, which poses significant difficulties. As a result, many industrial autonomous driving solutions still prefer pillar-based solutions [19], which flatten LiDAR points and process them with a 2D CNN. However, these approaches cannot take full advantage of 3D geometry and could be very slow when generalizing to perception ranges in real applications (*e.g*. >500m on the highway).

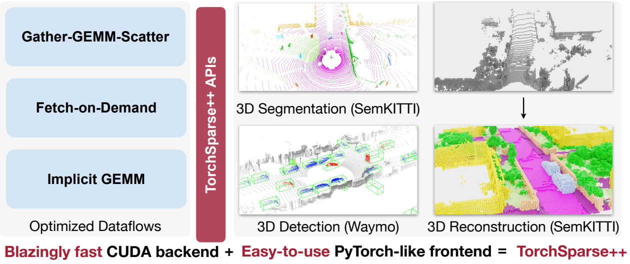

Gather-GEMM-Scatter
Fetch-on-Demand
Implicit GEMM
Optimized Dataflows
TorchSparse++ APIs
3D Segmentation (SemKITTI)
3D Detection (Waymo)
3D Reconstruction (SemKITTI)
Blazingly fast CUDA backend + Easy-to-use PyTorch-like frontend = TorchSparse++

Figure 1. TorchSparse++ is a high-performance GPU library that provides highly-optimized dataflows for convolution on point clouds. It provides state-of-the-art inference and training performance for all driving-related applications (*e.g*. 3D semantic segmentation, object detection and scene reconstruction). Scenes source: [2, 9, 27].

Several pioneering implementations of sparse convolution have adopted different dataflows for this operator. For instance, SparseConvNet [16] and SpConv [40] use the vanilla gather-GEMM-scatter dataflow, while MinkowskiEngine [12] proposes the fetch-on-demand dataflow. TorchSparse [35] optimizes the gather-scatter paradigm by fusing memory operations and grouping computations adaptively into batches to improve device utilization. Recently, SpConv v2 [39, 40] has adapted the implicit GEMM dataflow for dense convolution to the sparse domain,

[*] indicates equal contributions. Contact: kentang@mit.edu.

achieving state-of-the-art performance on real-world workloads. However, all these libraries assume a single dataflow throughout the execution of the entire model, which limits the design space for kernel optimization, such as tiling parameters.

In this work, we present an in-depth analysis of existing dataflows. Based on our analysis, we identify significant room for optimization, even for well-engineered kernels tuned at the PTX assembly level in SpConv v2 [39, 40]. As an example, we could improve the balance between computation regularity and overhead, increase parallelism, and optimize kernel selection based on input characteristics.

We also extend the opset of existing inference engines to support all necessary primitives for auto-driving related applications, including 3D semantic segmentation, object detection, and scene reconstruction. Our system, TorchSparse++, has been evaluated on seven representative models across three benchmark datasets, achieving end-to-end speedups of **2.9**×, **3.3**×, **2.2**×, and **1.8**× on an NVIDIA A100 GPU, outperforming state-of-the-art systems such as MinkowskiEngine, SpConv 1.2, TorchSparse, and SpConv v2. Code is released at `https://github.com/mit-han-lab/torchsparse`.

2. Library

The goal of TorchSparse++ is to provide efficient system implementation for existing point cloud deep learning workloads in autonomous driving and allow the users to easily extend its support for emerging operators. To achieve this goal, we abstract point clouds as *sparse tensors*.

2.1. Overview

A point cloud sparse tensor can be defined as an unordered set of points with features: $\{(\boldsymbol{p}_j, \boldsymbol{x}_j)\}$. $\boldsymbol{p}_j$ is the quantized coordinates for the j^{th} point in the D-dimensional space $\mathbb{Z}^D$, and $\boldsymbol{x}_j$ is its C-dimensional feature vector in $\mathbb{R}^C$. In autonomous driving applications, we have $D = 3$, corresponding to 3D points from the LiDAR sensors. Coordinate quantization is done through $\boldsymbol{p} = \lfloor \boldsymbol{p}_i^{(\text{raw})} / \boldsymbol{v} \rfloor$, where $\boldsymbol{v}$ is the voxel size vector. `Unique` operation is further applied to all quantized coordinates. For example, in CenterPoint [44], the point clouds on Waymo [33] are quantized using $\boldsymbol{v}$ = [0.1m, 0.1m, 0.15m]. This means that we will only keep one point within each 0.1m×0.1m×0.15m grid.

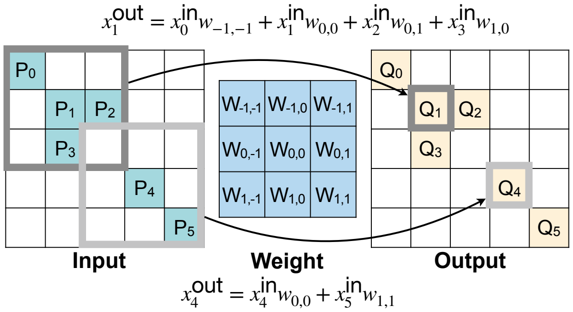

$x_1^{\text{out}} = x_0^{\text{in}} w_{-1,-1} + x_1^{\text{in}} w_{0,0} + x_2^{\text{in}} w_{0,1} + x_3^{\text{in}} w_{1,0}$
P_0
P_1
P_2
P_3
P_4
P_5
$W_{-1,-1}$
$W_{-1,0}$
$W_{-1,1}$
$W_{0,-1}$
$W_{0,0}$
$W_{0,1}$
$W_{1,-1}$
$W_{1,0}$
$W_{1,1}$
Q_0
Q_1
Q_2
Q_3
Q_4
Q_5
Input
Weight
Output
$x_4^{\text{out}} = x_4^{\text{in}} w_{0,0} + x_5^{\text{in}} w_{1,1}$

Figure 2. Sparse convolution (Equation 1) on $\Delta^2(3)$: computation is performed only on *nonzero* inputs.

Sparse Convolution In autonomous driving applications, the most important computation primitive in point cloud deep learning is sparse convolution [12, 16, 40]. Following the notations in [35], we define the D-dimensional neighborhood with kernel size K as $\Delta^D(K)$ (*e.g.* $\Delta^2(5)$ = $\{-2, -1, 0, 1, 2\}^2$ and $\Delta^3(3) = \{-1, 0, 1\}^3$). The forward form of sparse convolution (Figure 2) on the k^{th} output point is defined as:

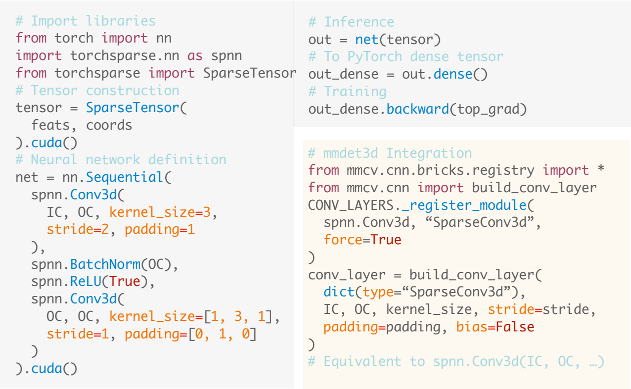
$$\boldsymbol{x}_k^{\text{out}} = \sum_{\boldsymbol{\delta} \in \boldsymbol{\Delta}^D(K)} \sum_j 1[\boldsymbol{p}_j = s\boldsymbol{q}_k + \boldsymbol{\delta}] \, (\boldsymbol{x}_j^{\text{in}} \cdot \boldsymbol{W}_{\boldsymbol{\delta}}), \qquad (1)$$

where $\boldsymbol{p}_j \in \boldsymbol{P}^{\text{in}}, \boldsymbol{q}_k \in \boldsymbol{P}^{\text{out}}$, $1[\cdot]$ is a binary indicator, s is the stride and $\boldsymbol{W}_\delta \in \mathbb{R}^{C_{\text{in}} \times C_{\text{out}}}$ corresponds to the weight matrix for kernel offset $\delta \in \Delta^D(K)$. Sparse convolution is closely connected to different convolutional primitives on point clouds. For example, PointNet++ [26] (used by PointRCNN [32], PV-RCNN [30, 31], 3D-SSD [41]) replaces the cubical neighborhood in Equation 1 with the k-nearest neighbor and performs reduction using max pooling instead of summation.

Other Functions TorchSparse++ leverages PyTorch's elementwise deep learning primitives, such as normalization layers and activation functions, since these non-spatial layers do not require dedicated sparsity support. Additionally, we have implemented the sparse counterpart of pooling layers. In contrast to conventional dense workloads, where the mapping between logical coordinates and physical storage location is straightforward (e.g., on an $H \times W$ image, pixel (h, w) is stored at memory location $hW + w$), such mapping does not hold for sparse point clouds. To address this issue, we provide a parallel hashtable abstraction in TorchSparse++, which supports fast coordinate-to-memory location queries.

2.2. APIs and Integration

APIs. TorchSparse++ offers easy-to-use and PyTorch-like APIs, depicted in Figure 3 and Figure 4. Both `module` and `functional` implementations are provided for all sparse point cloud operators. To construct a sparse convolutional network for point clouds, only a conversion from PyTorch's `nn.Conv3d`, `nn.BatchNorm3d`, and `nn.ReLU` to our `spnn` counterpart is required, as shown in Figure 3. Unlike other libraries, users do not need to specify any additional fields at module initialization time in TorchSparse++. For example, SpConv requires an `indice_key` field during convolution class initialization to help the system exploit map reuse

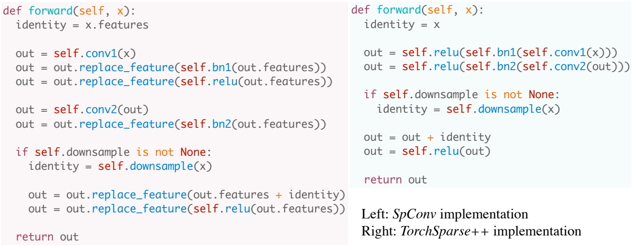


```
# Import libraries
from torch import nn
import torchsparse.nn as spnn
from torchsparse import SparseTensor
# Tensor construction
tensor = SparseTensor(
  feats, coords
).cuda()
# Neural network definition
net = nn.Sequential(
  spnn.Conv3d(
    IC, OC, kernel_size=3,
    stride=2, padding=1
  ),
  spnn.BatchNorm(OC),
  spnn.ReLU(True),
  spnn.Conv3d(
    OC, OC, kernel_size=[1, 3, 1],
    stride=1, padding=[0, 1, 0]
  )
).cuda()
```

```
# Inference
out = net(tensor)
# To PyTorch dense tensor
out_dense = out.dense()
# Training
out_dense.backward(top_grad)
```

```
# mmdet3d Integration
from mmcv.cnn.bricks.registry import *
from mmcv.cnn import build_conv_layer
CONV_LAYERS._register_module(
  spnn.Conv3d, “SparseConv3d”,
  force=True
)
conv_layer = build_conv_layer(
  dict(type=“SparseConv3d”),
  IC, OC, kernel_size, stride=stride,
  padding=padding, bias=False
)
# Equivalent to spnn.Conv3d(IC, OC, …)
```

Figure 3. TorchSparse++ offers user-friendly, PyTorch-like interfaces that allow seamless support of both training and inference. It can also be easily integrated into open-source frameworks such as mmdet3d [13] with less than 10 lines of code.



```
def forward(self, x):
  identity = x.features

  out = self.conv1(x)
  out = out.replace_feature(self.bn1(out.features))
  out = out.replace_feature(self.relu(out.features))

  out = self.conv2(out)
  out = out.replace_feature(self.bn2(out.features))

  if self.downsample is not None:
    identity = self.downsample(x)

    out = out.replace_feature(out.features + identity)
    out = out.replace_feature(self.relu(out.features))

  return out
```

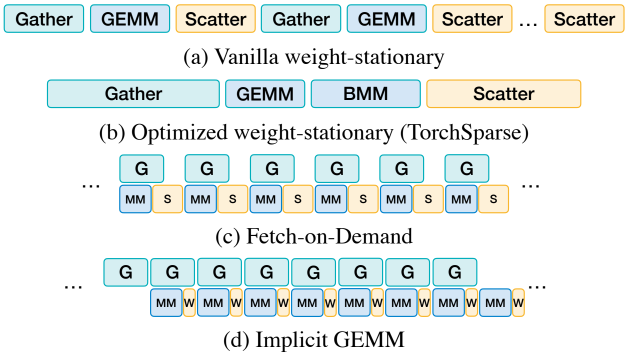
```
def forward(self, x):
  identity = x

  out = self.relu(self.bn1(self.conv1(x)))
  out = self.relu(self.bn2(self.conv2(out)))

  if self.downsample is not None:
    identity = self.downsample(x)

  out = out + identity
  out = self.relu(out)

  return out
```

Left: *SpConv* implementation
Right: *TorchSparse++* implementation

Figure 4. TorchSparse++ offers a much more intuitive implementation for a residual block composed of sparse convolution layers compared with SpConv [39, 40].

opportunities. However, this is done automatically in the TorchSparse++ frontend without any user annotation. We also provide exactly the same forward and backward implementation conventions as PyTorch. As in Figure 4, it is much more intuitive in TorchSparse++ to implement a sparse residual block compared with SpConv. Furthermore, when integrated into existing frameworks such as mmdet3d [13], who provides existing implementations for the residual block (and many other Conv2d-based blocks) for images, one only needs to override its member modules with spnn equivalence (*e.g*. replacing `nn.ReLU` with `spnn.ReLU`) and does not even need to write the forward function. Our API design greatly simplifies the development of point cloud models.

Integration. It is easy to integrate TorchSparse++ into existing algorithm frameworks and open-source repositories. For example, as in Figure 3, frameworks like mmdet3d [13] often provide *registries* to support building modules from a configuration dictionary. Registering operators in TorchSparse++ is as simple as registering a `Conv2d` layer to these frameworks. Users can simply modify the `type` field in the configuration dictionary passed to `build_conv_layer` from `Conv2d` to `SparseConv3d` to switch between the two layer types. All other layer parameters, such as input and output channels, kernel size, stride, padding, and bias, retain their equivalent meanings to their PyTorch 2D counterpart. We provide an example of open-source integration of TorchSparse++ in `https://github.com/mit-han-lab/bevfusion`, where our system serves as the backend for the LiDAR backbone in a multi-modality 3D perception model [23].



(a) Vanilla weight-stationary

(b) Optimized weight-stationary (TorchSparse)

(c) Fetch-on-Demand

(d) Implicit GEMM

Figure 5. Waterfall diagram for different dataflows for sparse convolution on GPU: weight-stationary dataflows (a, b) are easier to implement and maintain but they do not overlap memory access with computation. Both fetch-on-demand and implicit GEMM dataflows require custom MMA routines but are able to hide the memory access time with pipelining.

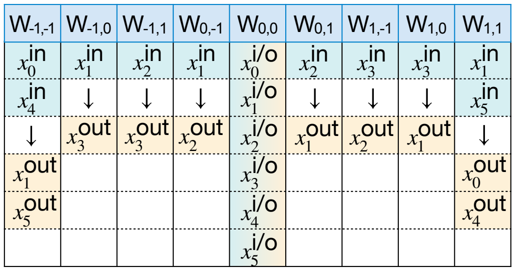
It is also extremely simple to implement new 3D deep learning models from scratch using TorchSparse++. We provide another example, available at `https://github.com/mit-han-lab/spvnas`, which demonstrates how our system can be applied to a complex 3D semantic segmentation model [36]. This model features custom voxelization/devoxelization operators and supports neural architecture search. Other existing implementations such as MinkowskiEngine and SpConv v2 could not provide such flexibility to support this model.

3. Implementation

Although most functions in TorchSparse++ are implemented straightforwardly, efficiently mapping the sparse convolution operator onto GPUs poses a nontrivial challenge. In this section, we describe three alternative implementations for sparse convolution that we have developed to address this challenge. Our approaches significantly improve upon existing implementations by introducing tensor core intrinsics, enhancing computation regularity and parallelism.

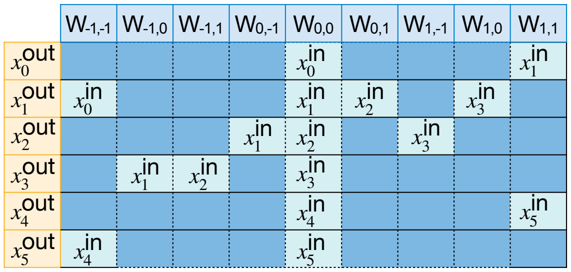
3.1. Gather-GEMM-Scatter Dataflow

Overview. A gather-GEMM-scatter [16, 40] implementation of sparse convolution (Figure 6) will finish all computation for one weight $\boldsymbol{W}_\delta$ before moving on to the other. It has an outmost host loop over K^D kernel offsets. For each offset δ, we first calculate all pairs $\mathcal{M}_\delta = \{(\boldsymbol{p}_j, \boldsymbol{q}_k)\}$ such that $\boldsymbol{p}_j = s\boldsymbol{q}_k + \boldsymbol{\delta}$. As is shown in Figure 5a, we then group all input features $\{\boldsymbol{x}_j^{\text{in}}\}$ together, resulting in a



$W_{-1,-1}$	$W_{-1,0}$	$W_{-1,1}$	$W_{0,-1}$	$W_{0,0}$	$W_{0,1}$	$W_{1,-1}$	$W_{1,0}$	$W_{1,1}$
x_0^{in}	x_1^{in}	x_2^{in}	x_1^{in}	$x_0^{i/o}$	x_2^{in}	x_3^{in}	x_3^{in}	x_1^{in}
x_4^{in}	↓	↓	↓	$x_1^{i/o}$	↓	↓	↓	x_5^{in}
↓	x_3^{out}	x_3^{out}	x_2^{out}	$x_2^{i/o}$	x_1^{out}	x_2^{out}	x_1^{out}	↓
x_1^{out}				$x_3^{i/o}$				x_0^{out}
x_5^{out}				$x_4^{i/o}$				x_4^{out}
				$x_5^{i/o}$				

Figure 6. Illustration of the gather-GEMM-scatter dataflow for Figure 2 workload: we first gather input features according to $\mathcal{M}_\delta$ for each weight δ, then perform GEMM or batched GEMM, and finally scatter the results back to output locations given in $\mathcal{M}_\delta$.



	$W_{-1,-1}$	$W_{-1,0}$	$W_{-1,1}$	$W_{0,-1}$	$W_{0,0}$	$W_{0,1}$	$W_{1,-1}$	$W_{1,0}$	$W_{1,1}$
x_0^{out}	🟦	🟦	🟦	🟦	x_0^{in}	🟦	🟦	🟦	x_1^{in}
x_1^{out}	x_0^{in}	🟦	🟦	🟦	x_1^{in}	x_2^{in}	🟦	x_3^{in}	🟦
x_2^{out}	🟦	🟦	🟦	x_1^{in}	x_2^{in}	🟦	x_3^{in}	🟦	🟦
x_3^{out}	🟦	x_1^{in}	x_2^{in}	🟦	x_3^{in}	🟦	🟦	🟦	🟦
x_4^{out}	🟦	🟦	🟦	🟦	x_4^{in}	🟦	🟦	🟦	x_5^{in}
x_5^{out}	x_4^{in}	🟦	🟦	🟦	x_5^{in}	🟦	🟦	🟦	🟦

Figure 7. Illustration of the vanilla implicit GEMM dataflow for Figure 2 workload: each green grid corresponds to a C_{in}-dimensional input feature and blue grids correspond to redundant computation. The input feature matrix is **not** stored in DRAM. We assume that each thread block contains 3 threads (3 rows).

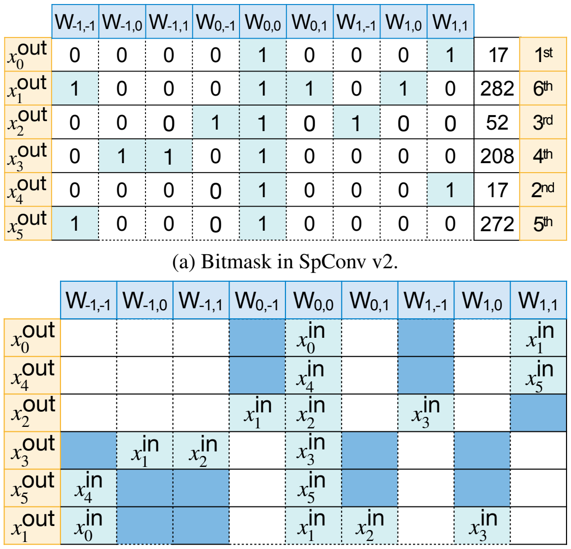
$|\mathcal{M}_\delta| \times C_{\text{in}}$ matrix in DRAM, and multiply it with weight $\boldsymbol{W}_\delta \in \mathbb{R}^{C_{\text{in}} \times C_{\text{out}}}$, finally we scatter the results back to output positions $\{\boldsymbol{x}_k^{\text{out}}\}$ according to $\mathcal{M}_\delta$. For example, since $\boldsymbol{p}_0 = 1 \times \boldsymbol{q}_1 + (-1,-1)$, $\boldsymbol{p}_4 = 1 \times \boldsymbol{q}_5 + (-1,-1)$, we group features $\boldsymbol{x}_0^{\text{in}}, \boldsymbol{x}_4^{\text{in}}$ together, multiply them by $\boldsymbol{W}_{-1,-1}$ and scatter back to outputs $\boldsymbol{x}_1^{\text{out}}, \boldsymbol{x}_5^{\text{out}}$.

Advantages. The gather-GEMM-scatter dataflow has small and controllable computation overhead, and is thus suitable for devices with limited computation capability. It is also easy to implement and maintain. After feature gathering, the main computation for each offset δ is simply dense matrix multiplication, and can be delegated to existing vendor libraries such as cuBLAS and cuDNN. As such, only the data movement operations (*i.e.* scatter and gather) need to be implemented and optimized in CUDA.

3.2. Fetch-on-Demand Dataflow

Overview. The gather-GEMM-scatter implementation requires three separate CUDA kernel calls in each host loop iteration over $\boldsymbol{\delta}$. An alternative fetch-on-demand dataflow [12, 17] merges the gather, matrix multiplication and scatter kernel calls into a single CUDA kernel. Instead of materializing the $|\mathcal{M}_\delta| \times C_{\text{in}}$ gather buffer in DRAM, it fetches $\{\boldsymbol{x}_j^{\text{in}} | (\boldsymbol{p}_j, \boldsymbol{q}_k) \in \mathcal{M}_\delta\}$ on demand into the L1 shared memory, performs matrix multiplication in the on-chip storage and directly scatters the partial sums (resided in the register file) to corresponding outputs $\{\boldsymbol{x}_k^{\text{out}} | (\boldsymbol{p}_j, \boldsymbol{q}_k) \in \mathcal{M}_\delta\}$ without first instantiating them in a DRAM scatter buffer.

Advantages. The fetch-on-demand dataflow enjoys similar benefit of low redundant computation to gather-GEMM-scatter. Despite not being able to exploit perfect reuse opportunities in both gathering and scattering as [35], it overlaps the computation with memory access operations. It also saves DRAM writes to large gather/scatter buffers. Thus, it is faster than a vanilla gather-scatter dataflow (Figure 5a).



	$W_{-1,-1}$	$W_{-1,0}$	$W_{-1,1}$	$W_{0,-1}$	$W_{0,0}$	$W_{0,1}$	$W_{1,-1}$	$W_{1,0}$	$W_{1,1}$		
x_0^{out}	0	0	0	0	1	0	0	0	1	17	1st
x_1^{out}	1	0	0	0	1	1	0	1	0	282	6th
x_2^{out}	0	0	0	1	1	0	1	0	0	52	3rd
x_3^{out}	0	1	1	0	1	0	0	0	0	208	4th
x_4^{out}	0	0	0	0	1	0	0	0	1	17	2nd
x_5^{out}	1	0	0	0	1	0	0	0	0	272	5th

(a) Bitmask in SpConv v2.

	$W_{-1,-1}$	$W_{-1,0}$	$W_{-1,1}$	$W_{0,-1}$	$W_{0,0}$	$W_{0,1}$	$W_{1,-1}$	$W_{1,0}$	$W_{1,1}$
x_0^{out}				🟦	x_0^{in}		🟦		x_1^{in}
x_4^{out}				🟦	x_4^{in}		🟦		x_5^{in}
x_2^{out}				x_1^{in}	x_2^{in}		x_3^{in}		🟦
x_3^{out}	🟦	x_1^{in}	x_2^{in}		x_3^{in}	🟦		🟦	
x_5^{out}	x_4^{in}	🟦	🟦		x_5^{in}	🟦		🟦	
x_1^{out}	x_0^{in}	🟦	🟦		x_1^{in}	x_2^{in}		x_3^{in}	

(b) Sorting reduces redundant computation in SpConv v2.

Figure 8. SpConv v2 sorts the input bitmasks and reorders the computation accordingly. White grids correspond to skipped zero computation. As a result, redundant computation is reduced from 38 MACs (Figure 7) to 14 MACs for the example in Figure 2. For simplicity, we do not visualize mask splitting in SpConv v2. Each thread block has 3 threads.

3.3. Implicit GEMM Dataflow

Overview. A very recent advance in sparse convolution inference, SpConv v2 [39, 40], takes an alternative approach of implicit GEMM computation, which is originally designed for dense convolution on images in CUTLASS [18], cuDNN [11] and has been implemented in PointAcc [21], a specialized accelerator for point cloud workloads. As in Figure 7, the sparse convolution workload in Figure 2 is equivalent to a dense GEMM $\boldsymbol{C} = \boldsymbol{A}_{M\times K} \times \boldsymbol{B}_{K\times N}$ with $M = N_{\text{out}}$ (number of output points), $N = C_{\text{out}}, K = |\Delta^D(K)|C_{\text{in}}$. We visualize matrix $\boldsymbol{A}$ in Figure 7, where we have $N_{\text{out}} \times |\Delta^D(K)|$ grids and each grid corresponds to

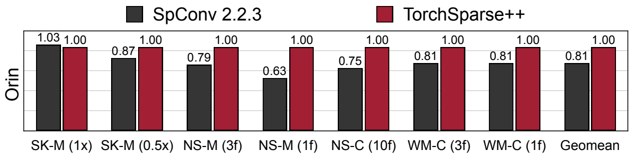


Figure 9. TorchSparse++ is **1.24**× faster on average and up to **1.57**× more efficient than SpConv 2.2.3 on resource-constrained NVIDIA Jetson Orin. All workloads run at >18 FPS, which is significantly faster than the LiDAR frequency.

C_{in} channels. Grids in green corresponds to input features. For example, output point $\boldsymbol{q}_1$ has four neighbors: $\boldsymbol{p}_0$ (with $\boldsymbol{w}_{-1,-1}$), $\boldsymbol{p}_1$ (with $\boldsymbol{w}_{0,0}$), $\boldsymbol{p}_2$ (with $\boldsymbol{w}_{0,1}$) and $\boldsymbol{p}_3$ (with $\boldsymbol{w}_{1,0}$) so the second row in Figure 7 has four green entries. *Implicit* GEMM means that instead of explicitly writing $\boldsymbol{A}$ into DRAM in an im2col [11] manner, we load tiles of $\boldsymbol{A}$ into the on-chip SRAM on the fly. Indices in $\boldsymbol{A}$ are mapped to locations in the input feature map according to neighborhood relationships.

SpConv v2: Sorted Implicit GEMM. In the Figure 7 example, we assume that each warp has three threads, and each thread performs calculation on one output point. A vanilla implicit GEMM implementation performs 0×0 computation for positions without corresponding neighbors. As such, there are 16 effective MACs and 38 wasted MACs in Figure 7. In real-world workloads, the amount of redundant computation ranges from 30% to 300% of effective MACs.

Therefore, Yan *et al.* [39, 40] proposed SpConv v2 to reduce the proportion of wasted computation. As in Figure 8a, a bitmask is first calculated for each output point. It indicates whether a specific neighbor of each output point exists. The bitmask is then converted to a decimal number and sorted. As such, in Figure 8b, the redundant MACs is reduced from 38 to 14. Experiments on real-world LiDAR scans show that sorting the bitmask can typically bring about a 1.3× to 2.0× reduction in redundant computation. To further reduce wasted MACs, Yan *et al.* split the bitmask into two trunks and sort these two bitmasks separately. A SplitK reduction [18] followed by output reordering is performed to get the final results.

Advantages. An implicit GEMM dataflow does not require scattering since its writeback stage is output-stationary. Input feature gathering and matrix multiplication-accumulation (MMA) operations can also be pipelined because they are implemented in a single CUDA kernel. Consequently, as shown in Figure 5d, the ideal runtime of an output-stationary kernel is $\max(\texttt{time\_gather}, \texttt{time\_mma} + \texttt{time\_wb})$, as opposed to $\texttt{time\_gather} + \texttt{time\_mma} + \texttt{time\_scatter}$ in a gather-GEMM-scatter dataflow.

4. Evaluation

4.1. Setup

We build our TorchSparse++ on top of TorchSparse [35] and compare it with four state-of-the-art sparse convolution libraries MinkowskiEngine 0.5.4 [12], SpConv 1.2.1 [40], TorchSparse [35] (gather-GEMM-scatter) and SpConv 2.2.3 [40] (sorted implicit GEMM). All systems except SpConv 1.2.1 are integrated into PyTorch 1.12.0 with CUDA 11.7 and cuDNN 8.4.1. SpConv 1.2.1 is incompatible with PyTorch 1.12.0 so we use PyTorch 1.9.0 instead.

We follow TorchSparse [35] to evaluate all systems on seven representative 3D deep learning workloads: MinkUNet [12] (0.5×/1× width) on SemanticKITTI [2], MinkUNet (1 or 3 frames) on nuScenes-LiDARSeg [3], CenterPoint [44] (10 frames) on nuScenes detection and CenterPoint (1 or 3 frames) on Waymo Open Dataset [33]. For detection workloads (CenterPoint), *we only evaluate the runtime of SparseConv layers*.

4.2. Inference Speedup

We compare our results with the baseline design MinkowskiEngine, SpConv 1.2.1, TorchSparse and SpConv 2.2.3 in Figure 10. All evaluations are done in unit batch size. TorchSparse++ outperforms baseline systems consistently on Tesla A100, RTX 3090, RTX 2080Ti and GTX 1080Ti. It achieves **2.9-3.7**×, **3.2-3.3**×, **2.0-2.2**× and **1.4-1.8**× measured end-to-end speedup over the state-of-the-art MinkowskiEngine, SpConv 1.2.1, TorchSparse and SpConv 2.2.3, respectively on Ampere GPUs and is 1.2-1.6× faster than SpConv 2.2.3 on Turing and Pascal GPUs. In Figure 9, we also compare TorchSparse++ with SpConv 2.2.3 on NVIDIA Jetson Orin, an edge GPU platform widely deployed on real-world autonomous vehicles. Our TorchSparse++ is **1.24**× faster than SpConv 2.2.3, while achieving up to **1.57**× speedup on nuScenes (with 32-beam LiDAR scans). Notably, recent advances in point cloud transformers [25, 34, 37] often claim superior accuracy-latency tradeoffs over sparse convolutional backbones implemented with the SpConv 2.2.3 backend. With the much faster TorchSparse++ backend, we argue that sparse convolutional networks are still very competitive in runtime.

5. Related Work

Point Cloud Inference Engines. Researchers have extensively developed efficient inference engines for sparse convolution. SpConv [40] proposes grid-based map search and the gather-GEMM-scatter dataflow. SparseConvNet [16] proposes hashmap-based map search and is later significantly improved (in latency) by MinkowskiEngine, which also introduces a new fetch-on-demand dataflow that excels at small workloads and allows generalized sparse convolution on >3D point clouds and on arbitrary coordinates.

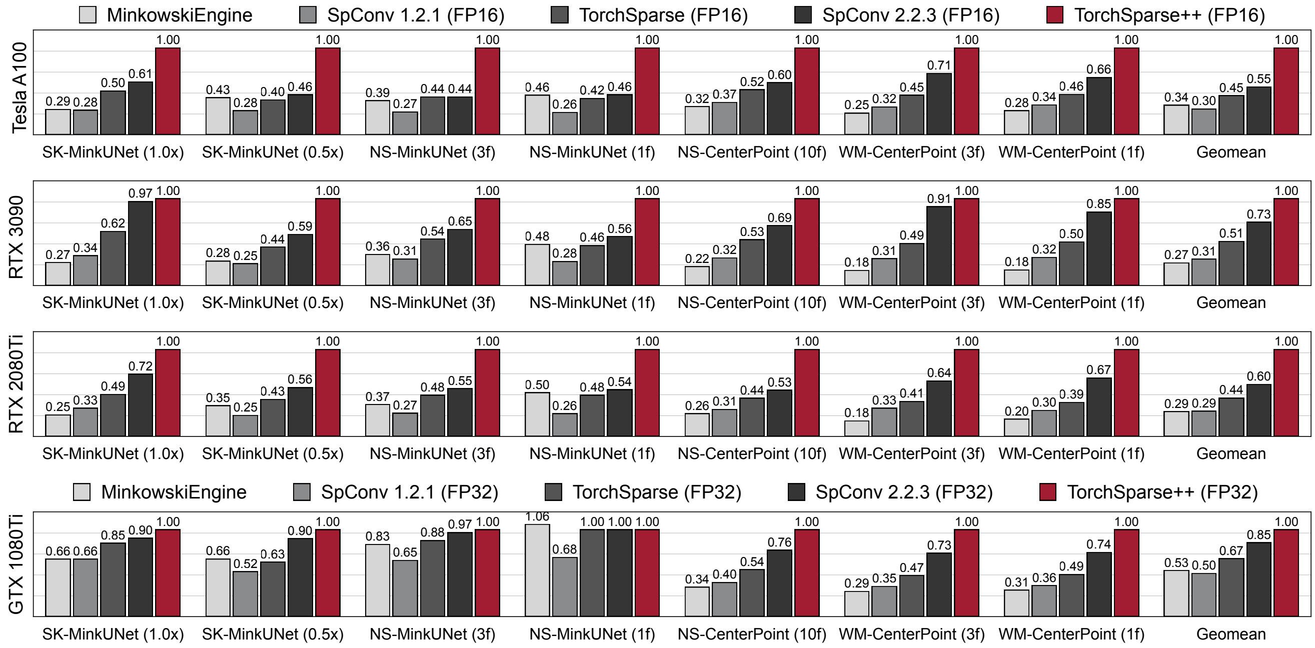


Figure 10. TorchSparse++ significantly outperforms existing point cloud inference engines in both 3D object detection and LiDAR segmentation benchmarks. It achieves **1.36-1.81**× geomean speedup over state-of-the-art SpConv 2.2.3 and is **1.96-2.20**× faster than TorchSparse on GPUs with the NVIDIA Ampere architecture. Furthermore, TorchSparse++ is 1.2-1.6× faster than SpConv 2.2.3 on Turing and Pascal architectures.

TorchSparse [35] pushes the performance of gather-GEMM-scatter by fusing memory operations together and adaptively grouping computation into batches. It also implements a GPU hash table and accelerates map search by exploiting kernel fusion and symmetry. More recently, SpConv v2 [40] switches to the sorted implicit GEMM workflow, inspired by CUTLASS [18]. It entirely rewrites CUTLASS for sparse workload and achieves remarkable speedup; however, it is engineering-expensive and hard to maintain. All the existing dataflows are further optimized in this work.

Tensor Program Optimization. TorchSparse++ is a system mainly constructed from hand-written CUDA kernels, but it can still benefit from recent advances in tensor program compilation and optimization. The pioneering research TVM [4] provides graph-level and operator-level abstractions for deep learning workloads based on the essence of Halide [28]. AutoTVM [5] proposes a learning-based, template-guided search framework to automatically discover the optimal mapping of a fixed-shape tensor program onto the target hardware. Nimble [29] and DietCode [45] are compilers based on TVM that can generate tensor programs with dynamic-shape workloads, but they are still tailored for dense workloads (*e.g*. transformers with variable length input sequences) and cannot deal with the sparsity in point clouds. More recently, TensorIR [14] proposes a new intermediate representation for tensor programs and allows easier tensorization of accelerator primitives. SparseTIR [43] further extends TensorIR to support sparse workloads. Bolt [38] combines fully-automatically generated kernels [4] with hand-written subroutines [18] via graph matching, achieving the best of both worlds. We expect that TorchSparse++ could further benefit from the recent progress in tensor program compilation and optimizations.

6. Conclusion

We introduce TorchSparse++, a high-performance GPU computation library designed for deep learning on point clouds. TorchSparse++ supports all primitives required for 3D semantic segmentation, object detection, and scene reconstruction workloads in autonomous driving. We conduct a holistic analysis of existing dataflows for point cloud convolution and further improved each dataflow to increase parallelism and balance control flow overhead and computation regularity. As a result, TorchSparse++ achieves impressive speedups, with 1.8-3.3× faster inference compared to state-of-the-art MinkowskEngine, SpConv v1/v2, and TorchSparse on seven real-world perception workloads. We hope that TorchSparse++ will facilitate research in 3D scene understanding for self-driving vehicles.

Acknowledgement. We would like to thank Yan Yan and Bohan Hou for helpful discussions. This work was supported by National Science Foundation, MIT-IBM Watson AI Lab, NVIDIA, Hyundai and Ford. Zhijian Liu was partially supported by the Qualcomm Innovation Fellowship.

References

[1] Xuyang Bai, Zeyu Hu, Xinge Zhu, Qingqiu Huang, Yilun Chen, Hongbo Fu, and Chiew-Lan Tai. TransFusion: Robust LiDAR-Camera Fusion for 3D Object Detection with Transformers. In *CVPR*, 2022. 1

[2] Jens Behley, Martin Garbade, Andres Milioto, Jan Quenzel, Sven Behnke, Cyrill Stachniss, and Juergen Gall. SemanticKITTI: A Dataset for Semantic Scene Understanding of LiDAR Sequences. In *IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019. 1, 5

[3] Holger Caesar, Varun Bankiti, Alex H. Lang, Sourabh Vora, Venice Erin Liong, Qiang Xu, Anush Krishnan, Yu Pan, Giancarlo Baldan, and Oscar Beijbom. nuScenes: A Multimodal Dataset for Autonomous Driving. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020. 5

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018. 6

[5] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to Optimize Tensor Programs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018. 6

[6] Xuesong Chen, Shaoshuai Shi, Benjin Zhu, Ka Chun Cheung, Hang Xu, and Hongsheng Li. MPPNet: Multi-Frame Feature Intertwining with Proxy Points for 3D Temporal Object Detection. In *ECCV*, 2022. 1

[7] Yukang Chen, Yanwei Li, Xiangyu Zhang, Jian Sun, and Jiaya Jia. Focal Sparse Convolutional Networks for 3D Object Detection. In *CVPR*, 2022. 1

[8] Yukang Chen, Jianhui Liu, Xiaojuan Qi, Xiangyu Zhang, Jian Sun, and Jiaya Jia. Scaling up Kernels in 3D CNNs. In *CVPR*, 2023. 1

[9] Ran Cheng, Christopher Agia, Yuan Ren, Xinhai Li, and Bingbing Liu. S3CNet: A Sparse Semantic Scene Completion Network for LiDAR Point Clouds. In *CoRL*, 2020. 1

[10] Ran Cheng, Ryan Razani, Ehsan Taghavi, Enxu Li, and Bingbing Liu. $(AF)^2$-S3Net: Attentive Feature Fusion with Adaptive Feature Selection for Sparse Semantic Segmentation Network. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021. 1

[11] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning. In *Advances in Neural Information Systems (NeurIPS) Workshops*, 2014. 4, 5

[12] Christopher Choy, JunYoung Gwak, and Silvio Savarese. 4D Spatio-Temporal ConvNets: Minkowski Convolutional Neural Networks. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019. 1, 2, 4, 5

[13] MMDetection3D Contributors. MMDetection3D: OpenMMLab next-generation platform for general 3D object detection. https://github.com/open-mmlab/mmdetection3d, 2020. 3

[14] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, and Tianqi Chen. TensorIR: An Abstraction for Automatic Tensorized Program Optimization. In *ASPLOS*, 2023. 6

[15] Runzhou Ge, Zhuangzhuang Ding, Yihan Hu, Wenxin Shao, Li Huang, Kun Li, and Qiang Liu. 1^{st} Place Solutions to the Real-time 3D Detection and the Most Efficient Model of the Waymo Open Dataset Challenge 2021. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2021. 1

[16] Benjamin Graham, Martin Engelcke, and Laurens van der Maaten. 3D Semantic Segmentation With Submanifold Sparse Convolutional Networks. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018. 1, 2, 3, 5

[17] Ke Hong, Zhongming Yu, Guohao Dai, Xinhao Yang, Yaoxiu Lian, Zehao Liu, Ningyi Xu, and Yu Wang. Exploiting Hardware Utilization and Adaptive Dataflow for Efficient Sparse Convolution in 3D Point Clouds. In *Sixth Conference on Machine Learning and Systems (MLSys)*, 2023. 4

[18] Andrew Kerr, Haicheng Wu, Manish Gupta, Dustyn Blasig, Pradeep Ramini, et al. CUTLASS: CUDA Template Library for Linear Algebra Subroutines. https://github.com/NVIDIA/CUTLASS, 2022. 4, 5, 6

[19] Alex H. Lang, Sourabh Vora, Holger Caesar, Lubing Zhou, and Jiong Yang. PointPillars: Fast Encoders for Object Detection from Point Clouds. In *CVPR*, 2019. 1

[20] Yanwei Li, Yilun Chen, Xiaojuan Qi, Zeming Li, Jian Sun, and Jiaya Jia. Unifying Voxel-based Representation with Transformer for 3D Object Detection. In *NeurIPS*, 2022. 1

[21] Yujun Lin, Zhekai Zhang, Haotian Tang, Hanrui Wang, and Song Han. PointAcc: Efficient Point Cloud Accelerator. In *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021. 4

[22] Zhijian Liu, Alexander Amini, Sibo Zhu, Sertac Karaman, Song Han, and Daniela Rus. Efficient and Robust LiDAR-Based End-to-End Navigation. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2021. 1

[23] Zhijian Liu, Haotian Tang, Alexander Amini, Xinyu Yang, Huizi Mao, Daniela Rus, and Song Han. BEVFusion: Multi-Task Multi-Sensor Fusion with Unified Bird's-Eye View Representation. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2023. 1, 3

[24] Zhijian Liu, Haotian Tang, Shengyu Zhao, Kevin Shao, and Song Han. PVNAS: 3D Neural Architecture Search with Point-Voxel Convolution. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 2021. 1

[25] Zhijian Liu, Xinyu Yang, Haotian Tang, Shang Yang, and Song Han. FlatFormer: Flattened Window Attention for Efficient Point Cloud Transformer. In *CVPR*, 2023. 5

[26] Charles Ruizhongtai Qi, Li Yi, Hao Su, and Leonidas J Guibas. PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017. 2

[27] Charles R Qi, Yin Zhou, Mahyar Najibi, Pei Sun, Khoa Vo, Boyang Deng, and Dragomir Anguelov. Offboard 3d object detection from point cloud sequences. In *CVPR*, 2021. 1

[28] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Fredo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *PLDI*, 2013. 6

[29] Haichen Shen, Jared Roesch, Zhi Chen, Wei Chen, Yong Wu, Mu Li, Vin Sharma, Zachary Tatlock, and Yida Wang. Nimble: Efficiently Compiling Dynamic Neural Networks for Model Inference. In *MLSys*, 2021. 6

[30] Shaoshuai Shi, Chaoxu Guo, Li Jiang, Zhe Wang, Jianping Shi, Xiaogang Wang, and Hongsheng Li. PV-RCNN: Point-Voxel Feature Set Abstraction for 3D Object Detection. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020. 2

[31] Shaoshuai Shi, Li Jiang, Jiajun Deng, Zhe Wang, Chaoxu Guo, Jinaping Shi, Xiaogang Wang, and Hongsheng Li. PV-RCNN++: Point-Voxel Feature Set Abstraction With Local Vector Representation for 3D Object Detection. *arXiv preprint arXiv:2102.00463*, 2021. 2

[32] Shaoshuai Shi, Xiaogang Wang, and Hongsheng Li. PointRCNN: 3D Object Proposal Generation and Detection From Point Cloud. In *CVPR*, 2019. 2

[33] Pei Sun, Henrik Kretzschmar, Xerxes Dotiwalla, Aurelien Chouard, Vijaysai Patnaik, Paul Tsui, James Guo, Yin Zhou, Yuning Chai, Benjamin Caine, Vijay Vasudevan, Wei Han, Jiquan Ngiam, Hang Zhao, Aleksei Timofeev, Scott Ettinger, Maxim Krivokon, Amy Gao, Aditya Joshi, Yu Zhang, Jonathon Shlens, Zhifeng Chen, and Dragomir Anguelov. Scalability in Perception for Autonomous Driving: Waymo Open Dataset. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020. 2, 5

[34] Pei Sun, Mingxing Tan, Weiyue Wang, Chenxi Liu, Fei Xia, Zhaoqi Leng, and Dragomir Anguelov. SWFormer: Sparse Window Transformer for 3D Object Detection in Point Clouds. In *ECCV*, 2022. 5

[35] Haotian Tang, Zhijian Liu, Xiuyu Li, Yujun Lin, and Song Han. TorchSparse: Efficient Point Cloud Inference Engine. In *Fifth Conference on Machine Learning and Systems (MLSys)*, 2022. 1, 2, 4, 5, 6

[36] Haotian Tang, Zhijian Liu, Shengyu Zhao, Yujun Lin, Ji Lin, Hanrui Wang, and Song Han. Searching Efficient 3D Architectures with Sparse Point-Voxel Convolution. In *European Conference on Computer Vision (ECCV)*, 2020. 1, 3

[37] Haiyang Wang, Chen Shi, Shaoshuai Shi, Meng Lei, Sen Wang, Di He, Bernet Schiele, and Liwei Wang. DSVT: Dynamic Sparse Voxel Transformer with Rotated Sets. In *CVPR*, 2023. 5

[38] Jiarong Xing, Leyuan Wang, Shang Zhang, Jack Chen, Ang Chen, and Yibo Zhu. Bolt: Bridging the Gap between Autotuners and Hardware-native Performance. In *MLSys*, 2022. 6

[39] Yan Yan. SpConv. https://github.com/traveller59/spconv, 2022. 1, 2, 3, 4, 5

[40] Yan Yan, Yuxing Mao, and Bo Li. SECOND: Sparsely Embedded Convolutional Detection. *Sensors*, 2018. 1, 2, 3, 4, 5, 6

[41] Zetong Yang, Yanan Sun, Shu Liu, and Jiaya Jia. 3DSSD: Point-Based 3D Single Stage Object Detector. *CVPR*, 2020. 2

[42] Dongqiangzi Ye, Weijia Chen, Zixiang Zhou, Yufei Xie, Yu Wang, Panqu Wang, and Hassan Foroosh. LidarMultiNet: Unifying LiDAR Semantic Segmentation, 3D Object Detection, and Panoptic Segmentation in a Single Multi-task Network. *arXiv*, 2022. 1

[43] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. SparseTIR: Composable Abstractions for Sparse Compilation in Deep Learning. In *ASPLOS*, 2023. 6

[44] Tianwei Yin, Xingyi Zhou, and Philipp Krähenbühl. Center-based 3D Object Detection and Tracking. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021. 1, 2, 5

[45] Bojian Zheng, Ziheng Jiang, Cody Hao Yu, Haichen Shen, Joshua Fromm, Yizhi Liu, Yida Wang, Luis Ceze, Tianqi Chen, and Gennady Pekhimenko. DietCode: Automatic Optimization for Dynamic Tensor Programs. In *MLSys*, 2022. 6

[46] Zixiang Zhou, Xiangchen Zhao, Yu Wang, Panqu Wang, and Hassan Foroosh. CenterFormer: Center-based Transformer for 3D Object Detection. In *ECCV*, 2022. 1