TorchSparse: Efficient Point Cloud Inference Engine

Haotian Tang\(^*\)\(^1\), Zhijian Liu\(^*\)\(^1\), Xiuyu Li\(^*\)\(^2\), Yujun Lin\(^1\), Song Han\(^1\)

Massachusetts Institute of Technology\(^1\) University of California, Berkeley\(^2\)

Project Page: https://torchsparse.mit.edu
Open Source: https://github.com/mit-han-lab/torchsparse
Point cloud data is everywhere

VR Glasses

Autonomous Driving Vehicles

AR iPhones and iPad

LiDAR Mapping Drones

Vacuum Cleaners

LiDAR Scanner
3D point cloud understanding is vital in auto-driving

Efficiency is crucial for automotive applications

3D Semantic Segmentation

The trunk of an auto-driving car

Full of computers!

Tang et al., Searching Efficient 3D Architectures with Sparse Point-Voxel Convolution, ECCV 2020.
3D point cloud understanding is vital in auto-driving

Efficiency is crucial for automotive applications

3D Object Detection

BEV Map Segmentation

Liu et al., BEVFusion: Multi-Task Multi-Sensor Fusion with Unified Bird’s-Eye-View Representation, arXiv 2022.
3D sparse CNNs are not well-optimized (yet)

More accurate, less computation, but slower!

Mean IoU

PolarNet (2D dense CNN) | 57.2 | 60.3
SPVNAS (3D sparse CNN)

#MACs (G)

PolarNet (2D dense CNN) | 135
SPVNAS (3D sparse CNN) | 9

Latency (ms)

PolarNet (2D dense CNN) | 62
SPVNAS (3D sparse CNN) | 89

Note: Mean IoU is measured on SemanticKITTI, while latency is measured on GTX 1080Ti GPU.
Sparse convolution is tailored for point clouds

Conventional Convolution

Input sparsity from ReLU
Nonzeros will dilate

Sparse Convolution

Input sparsity from the distribution in physical space
Nonzeros will not dilate

Graham, Submanifold Sparse Convolutional Neural Networks, BMVC 2015
Sparse convolution is inefficient on GPUs

Bottleneck is data movement and mapping due to sparsity and irregularity
Results for TorchSparse Optimizations

Trade computation for regularity and reduce memory footprint

- Data Movement
- MatMul
- Mapping
- Conv2D / NMS
- Misc

Segmentation (Before Opt.) vs (After Opt.):
- 1.7× faster

Detection (Before Opt.) vs (After Opt.):
- 2× faster
Background

Definition and existing implementation of sparse convolution
Sparse convolution computation

A sparse set of dense MMA, with rules defined by maps

Conventional Convolution

Sparse Convolution

Maps
(In, Out, Wgt)

Computation
($f_{out} = f_{out} + f_{in} \times W_{wgt}$) for each entry in the maps

$(P_0, Q_0, W_{1,1})$

No compute
Sparse convolution computation

A sparse set of dense MMA, with rules defined by maps

Conventional Convolution

Sparse Convolution

Maps
(In, Out, Wgt)

Computation
($f_{out} = f_{out} + f_{in} \times W_{wgt}$) for each entry in the maps

$(P_0, Q_0, W_{1,1})$
$(P_0, Q_1, W_{1,0})$

No compute
No compute
Sparse convolution computation

A sparse set of dense MMA, with rules defined by maps

Conventional Convolution

Sparse Convolution

Maps

\((\text{In}, \text{Out}, \text{Wgt})\)

Computation

\(f_{\text{out}} = f_{\text{out}} + f_{\text{in}} \times W_{\text{wgt}}\) for each entry in the maps
Sparse convolution computation

A sparse set of dense MMA, with rules defined by maps

Conventional Convolution

Sparse Convolution

Maps
(In, Out, Wgt)

(P₀, Q₀, W₁,₁)
(P₀, Q₁, W₁,₀)
(P₀, Q₂, W₁,₋₁)
(P₀, Q₃, W₀,₁)

Computation
(f_{out} = f_{out} + f_{in} × W_{Wgt}) for each entry in the maps

No compute
No compute
No compute
No compute
Sparse convolution computation

A *sparse* set of *dense* MMA, with rules defined by *maps*

**Conventional Convolution**

- $(P_0, Q_0, W_{1,1})$
- $(P_0, Q_1, W_{1,0})$
- $(P_0, Q_2, W_{1,-1})$
- $(P_0, Q_3, W_{0,1})$
- $(P_0, Q_4, W_{0,0})$

**Sparse Convolution**

- No compute
- No compute
- No compute
- No compute
- $(P_0, Q_0, W_{0,0})$

**Maps**

- $(\text{In}, \text{Out}, Wgt)$

**Computation**

- $(f_{\text{Out}} = f_{\text{Out}} + f_{\text{In}} \times W_{\text{Wgt}})$ for each entry in the maps
Sparse convolution computation

A sparse set of dense MMA, with rules defined by maps

Conventional Convolution

Sparse Convolution

Maps
(In, Out, Wgt)

Computation
($f_{out} = f_{out} + f_{in} \times W_{wgt}$) for each entry in the maps
Sparse convolution computation

A **sparse** set of **dense** MMA, with rules defined by **maps**

**Conventional Convolution**

**Sparse Convolution**

**Maps**
\[(\text{In}, \text{Out}, Wgt)\]

**Computation**
\[f_{\text{Out}} = f_{\text{Out}} + f_{\text{In}} \times W_{\text{Wgt}}\] for each entry in the maps

(P₀, Q₀, W₁₁)
(P₀, Q₁, W₁₁)
(P₀, Q₂, W₁₁)
(P₀, Q₃, W₀₁)
(P₀, Q₄, W₀₀)
(P₀, Q₅, W₀₋₁)
(P₀, Q₆, W₋₁₁)

No compute
No compute
No compute
No compute
(P₀, Q₀, W₀₀)
No compute
No compute
Sparse convolution computation

A sparse set of dense MMA, with rules defined by maps

Conventional Convolution

Sparse Convolution

Maps
(In, Out, Wgt)

Computation
\( f_{\text{Out}} = f_{\text{Out}} + f_{\text{In}} \times W_{\text{Wgt}} \) for each entry in the maps

(P₀, Q₀, W₁,₁)
(P₀, Q₁, W₁,₀)
(P₀, Q₂, W₁,⁻¹)
(P₀, Q₃, W₀,₁)
(P₀, Q₄, W₀,₀)
(P₀, Q₅, W₀,⁻¹)
(P₀, Q₆, W⁻¹,₁)
(P₀, Q₇, W⁻¹,₀)
(P₀, Q₈, W₀,₀)
(P₀, Q₉, W₋₁,₀)

No compute
No compute
No compute
No compute
No compute
No compute
No compute
No compute
No compute
No compute
Sparse convolution computation

A sparse set of dense MMA, with rules defined by maps

Conventional Convolution

Sparse Convolution

Maps
(In, Out, Wgt)

Computation
($f_{Out} = f_{Out} + f_{In} \times W_{Wgt}$) for each entry in the maps

9 matrix multiplications

2 matrix multiplications
Existing GPU implementation of sparse convolution

Weight-stationary computation, separate matmul for different weights

\[
\begin{align*}
f_1 &= f_1 + f_0 \times W_{-1,-1} \\
f_4 &= f_4 + f_3 \times W_{-1,-1}
\end{align*}
\]
Existing GPU implementation of sparse convolution

Weight-stationary computation, separate matmul for different weights

Input Features  | Input Buffer  | Weight | Partial Sum | Output Features
---|---|---|---|---
\(P_0\) | 5 \times C_{in} | \(1 \times C_{in}\) | \(C_{in} \times C_{out}\) | 1 \times C_{out} | \(Q_0\)
\(P_1\) | | | | | \(Q_1\)
\(P_2\) | | | | | \(Q_2\)
\(P_3\) | | | | | \(Q_3\)
\(P_4\) | | | | | \(Q_4\)

\(f_3 = f_3 + f_1 \times W_{-1,0}\)
Existing GPU implementation of sparse convolution

Weight-stationary computation, separate matmul for different weights

Note: maps for $W_{0,0}$ contains all entries.
Existing GPU implementation of sparse convolution

Weight-stationary computation, separate matmul for different weights

\[ f_1 = f_1 + f_3 \times W_{1,0} \]
Existing GPU implementation of sparse convolution

Weight-stationary computation, separate matmul for different weights

\[
\begin{align*}
\text{Input Features} & \rightarrow \text{Input Buffer} \rightarrow \text{Weight} \rightarrow \text{Partial Sum} \rightarrow \text{Output Features} \\
\text{Maps} & \quad \text{(In, Out, Wgt)} \\
(P_0, Q_1, W_{-1,-1}) & \\
(P_3, Q_4, W_{-1,-1}) & \\
(P_1, Q_3, W_{-1,0}) & \\
(P_0, Q_0, W_{0,0}) & \\
(P_1, Q_1, W_{0,0}) & \\
(P_2, Q_2, W_{0,0}) & \\
(P_3, Q_3, W_{0,0}) & \\
(P_4, Q_2, W_{0,0}) & \\
(P_3, Q_1, W_{1,0}) & \\
(P_1, Q_3, W_{1,1}) & \\
(P_4, Q_2, W_{1,1}) &
\end{align*}
\]

Workload

\[
\begin{align*}
 f_0 &= f_0 + f_1 \times W_{1,1} \\
 f_3 &= f_3 + f_4 \times W_{1,1}
\end{align*}
\]
TorchSparse Overview

System, API and optimization overview
TorchSparse: Efficient Point Cloud Inference Engine

torchsparse.mit.edu

Haotian Tang*, Zhijian Liu*, Xiuyu Li*, Yujun Lin and Song Han
TorchSparse has PyTorch-like APIs

```python
import torch.nn as nn

class ConvBlock(nn.Sequential):
    def __init__(self,
        in_channels: int,
        out_channels: int,
        kernel_size: Union[int, list, tuple],
        stride: Union[int, list, tuple] = 1,
        dilation: int = 1) -> None:
        super().__init__(
            nn.Conv2d(in_channels,
                       out_channels,
                       kernel_size,
                       stride=stride,
                       dilation=dilation),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(True))
```

```python
import torchsparse.nn as spnn

class SparseConvBlock(nn.Sequential):
    def __init__(self,
        in_channels: int,
        out_channels: int,
        kernel_size: Union[int, list, tuple],
        stride: Union[int, list, tuple] = 1,
        dilation: int = 1) -> None:
        super().__init__(
            spnn.Conv3d(in_channels,
                        out_channels,
                        kernel_size,
                        stride=stride,
                        dilation=dilation),
            spnn.BatchNorm(out_channels),
            spnn.ReLU(True))
```
TorchSparse optimization overview

Locality-Aware Access  Adaptive Grouping  Locality-Aware Access

Gather  Matrix-Matrix Multiplication  Scatter-Accumulate

Input Features  Apply BMM  Output Features

F0  F3  PSUM 1  PSUM 4
F1  pad  W_{-1,1}  =  PSUM 1  PSUM 4
F3  pad  W_{-1,0}  =  PSUM 3
F1  W_{1,0}  =  PSUM 1
F4  W_{1,1}  =  PSUM 0  PSUM 3
F0  W_{0,0}  =  PSUM 0  PSUM 1
F3  Apply MM  PSUM 3
F4  PSUM 4

Gather  Scatter  Adaptive Grouping  Scatter

Locality-Aware Access

Haotian Tang*, Zhijian Liu*, Xiuyu Li*, Yujun Lin and Song Han

TorchSparse: Efficient Point Cloud Inference Engine  torchsparse.mit.edu
Trading computation for regularity

Optimizing matrix multiplication via adaptive grouping

Matrix-Matrix Multiplication

Gather

Adaptive Grouping

Locality-Aware Access

Input Features

F0
F1
F2
F3
F4

Gather

F0
F3
F1
F3
pad
F1
F4
F0
F1
F2
F3
F4

W_{-1,-1} \times F0 = PSUM_{1}
W_{-1,0} \times F1 = PSUM_{3}
W_{1,0} \times F3 = PSUM_{1}
W_{1,1} \times F4 = PSUM_{3}

Apply BMM

W_{0,0} \times F0 = PSUM_{0}
W_{0,1} \times F1 = PSUM_{1}
W_{1,0} \times F2 = PSUM_{2}
W_{1,1} \times F3 = PSUM_{3}
W_{1,2} \times F4 = PSUM_{4}

Apply MM

Scatter

Output Features

F0
F1
F2
F3
F4
Reducing memory footprint

Optimizing data movement with fused and locality-aware memory access
Trading computation for regularity

Matrix multiplication optimizations
Trading computation for regularity

Separate computation (baseline): many kernel calls, low device utilization

Separate Computation

Worst

Best

Computation overhead

Computation regularity
Trading computation for regularity

Dense convolution: best regularity but large computation overhead

Separate Computation

Worst

Best

Computation overhead

Computation regularity

Dense Convolution

Worst

Best

Computation overhead

Computation regularity
Trading computation for regularity

Computation with grouping: balancing overhead and regularity

Separate Computation

Worst

Best

Computation overhead

Computation regularity

Dense Convolution

Worst

Best

Computation overhead

Computation regularity

Computation with grouping

Worst

Best

Extra computation = 2 / 28 (Small overhead)

Computation overhead

Computation regularity
Trading computation for regularity

Searching customized strategy for different model and datasets

Increasing regularity helps improve latency

Padding overhead hurts latency

Number of Groups

Speedup Over Baseline

SemanticKITTI

nuScenes

Map Size

Weight Index

1 4 7 10 13 16 19 22 2527

10^2 10^3 10^4 10^5

1 4 7 10 13 16 19 22 2527

10^2 10^3 10^4 10^5
Results on matrix multiplication optimizations

SemanticKITTI

| TFLOP/s | Baseline | Fixed Grouping | Adaptive Grouping |
|---------|----------|----------------|-------------------|
| 8.1     | 8.7      | 11.9           |                   |
| Normalized Speedup | 1.00 | 0.87 | 1.39 |

**TorchSparse**: Efficient Point Cloud Inference Engine

torchsparse.mit.edu

Haotian Tang*, Zhijian Liu*, Xiuyu Li*, Yujun Lin and Song Han
Results on matrix multiplication optimizations

nuScenes: fixed grouping has best TFLOP/s but adaptive grouping is faster

This is because fixed grouping introduced large amount of redundant computation.
Reducing memory footprint

Data movement optimizations
Recall: data movement is a major overhead

43-47% of total runtime for 3D perception models

- Data Movement: 5%
- GEMM: 44%
- Mapping: 4%
- 2D/NMS: 47%
- Misc.: 4%

Semantic Segmentation

Object Detection
Quantized and vectorized memory access reduces the memory footprint by 2x.
Weight-stationary scatter-gather: not cache friendly

Cache Hit

\[(P_0, Q_1) \rightarrow (P_3, Q_4) \rightarrow \ldots \rightarrow (P_{95029}, Q_{95133}) \rightarrow (P_{95077}, Q_{95180}) \rightarrow (P_{95133}, Q_{95229}) \rightarrow W_{-1,-1,-1} \]

Cache Miss

\[(P_1, Q_0) \rightarrow (P_4, Q_2) \rightarrow \ldots \rightarrow (P_{95133}, Q_{95029}) \rightarrow (P_{95180}, Q_{95077}) \rightarrow (P_{95229}, Q_{95133}) \rightarrow W_{1,1,1} \]
Weight-stationary scatter-gather: not cache friendly

Gather phase for weight $W_{-1,-1,-1}$

Cache Hit

Cache Miss

$(P_0, Q_1)$

$W_{-1,-1,-1}$  $W_{-1,-1,0}$  $W_{1,1,0}$  $W_{1,1,1}$
Weight-stationary scatter-gather: not cache friendly

Gather phase for weight \( W_{-1,-1,-1} \)

\[
(P_0, Q_1) \\
(P_3, Q_4)
\]

\[
W_{-1,-1,-1} \\
W_{-1,-1,0} \\
W_{1,1,0} \\
W_{1,1,1}
\]
Gather phase for weight $W_{-1,-1,-1}$

Because the maps for $W_{-1,-1,-1}$ are unique, there is no cache reuse during the gather phase for $W_{-1,-1,-1}$.
Weight-stationary scatter-gather: not cache friendly

Scatter phase for weight $W_{-1,-1,-1}$

Cache Hit

Cache Miss

$W_{-1,-1,-1}$  $W_{-1,-1,0}$  $W_{1,1,0}$  $W_{1,1,1}$
Weight-stationary scatter-gather: not cache friendly

Scatter phase for weight $W_{-1,-1,-1}$

\[
\begin{align*}
(P_0, Q_1) & \rightarrow \ldots \\
(P_3, Q_4) & \rightarrow \ldots \\
(P_{95029}, Q_{95133}) & \rightarrow \ldots \\
(P_{95077}, Q_{95181}) & \rightarrow \ldots
\end{align*}
\]

$W_{-1,-1,-1}$ $W_{-1,-1,0}$ $W_{1,1,0}$ $W_{1,1,1}$
Weight-stationary scatter-gather: not cache friendly

Scatter phase for weight $W_{-1,-1,-1}$

Because the maps for $W_{-1,-1,-1}$ are unique, there is no cache reuse during the scatter phase for $W_{-1,-1,-1}$
Weight-stationary scatter-gather: not cache friendly

It is also the case for all other weights because the cache is not large enough to hold all input features / output partial sums.
Solution: fused and locality-aware scatter-gather

Changing the map layout

Weight-stationary: vertical traversal
For each weight, what are the (input, output) pairs using this weight?

Input/Output-stationary: horizontal traversal
For each input/output, what weight sub-map does it appear in?
Solution: fused and locality-aware scatter-gather

Gather phase for all input points

Input-stationary gather

First accesses to all input points are mandatory cache misses.
Solution: fused and locality-aware scatter-gather

Gather phase for all input points

| Cache Hit | Cache Miss |
|-----------|------------|

P₀ \(W_{-1,-1,-1} \rightarrow W_{-1,1,0} \rightarrow W_{1,0,1}\)  

P₁ \(W_{1,0,1} \rightarrow W_{1,1,1}\)  
...  

P₉₅₀₇ \(W_{-1,-1,-1} \rightarrow W_{-1,-1,0}\)  
...  

P₉₅₁₃ \(W_{-1,-1,0} \rightarrow W_{1,1,1}\)  
...  

P₉₅₂₂₉ \(W_{1,1,0}\)  

**Input-stationary gather**

All subsequent accesses **hits** because the loaded input features are stored in the **register file**.
Solution: fused and locality-aware scatter-gather

Scatter phase for all output points

**Input-stationary** gather

**Output-stationary** scatter

The situation is similar for output-stationary scatter.
Solution: fused and locality-aware scatter-gather

Improving the cache hit ratio via reordering memory accesses

(a) baseline: weight-stationary scatter-gather

(b) ours: fused and locality-aware scatter-gather
Evaluation
TorchSparse achieves consistent improvements on different GPUs

| Baseline Implementation | MinkowskiEngine 0.5.4 | SPConv 1.2.1 (FP32) | SPConv 1.2.1 (FP16) | TorchSparse |
|-------------------------|-----------------------|---------------------|---------------------|-------------|
| SK-MinkUNet (1.0x)      | 1.00                  | 0.65                | 0.73                | 0.59        |
| SK-MinkUNet (0.5x)      | 0.660.67               | 1.00                | 0.550.55            | 0.59        |
| NS-MinkUNet (3f)        | 0.660.73               | 0.75                | 0.780.81            | 0.65        |
| NS-MinkUNet (1f)        | 1.00                  | 0.550.57            | 1.00                | 0.65        |
| NS-CenterPoint (10f)    | 0.67                  | 0.60.64             | 0.85                | 0.65        |
| WM-CenterPoint (3f)     | 0.60.64               | 0.60.54             | 0.83                | 0.65        |
| WM-CenterPoint (1f)     | 1.00                  | 1.00                | 0.83                | 0.65        |
| Geomean                 | 0.650.65              | 0.650.65            | 0.650.65            | 0.65        |

SK: SemanticKITTI (ICCV’19), NS: nuScenes (CVPR’20), WM: Waymo (CVPR’20)
TorchSparse achieves consistent improvements on different GPUs

|                      | RTX 3090 | RTX 2080Ti |
|----------------------|----------|------------|
| **Baseline Implementation** | 1.00     | 1.00       |
| **MinkowskiEngine 0.5.4** | 1.00     | 1.00       |
| **SPConv 1.2.1 (FP32)** | 1.00     | 1.00       |
| **SPConv 1.2.1 (FP16)** | 1.00     | 1.00       |
| **TorchSparse**        | 1.00     | 1.00       |

- **SK-MinkUNet (1.0x)**: SemanticKITTI (ICCV’19)
- **SK-MinkUNet (0.5x)**: SemanticKITTI (ICCV’19)
- **NS-MinkUNet (3f)**: nuScenes (CVPR’20)
- **NS-MinkUNet (1f)**: nuScenes (CVPR’20)
- **NS-CenterPoint (10f)**: Waymo (CVPR’20)
- **WM-CenterPoint (3f)**: Waymo (CVPR’20)
- **WM-CenterPoint (1f)**: Waymo (CVPR’20)
- **Geomean**
### Results

TorchSparse achieves consistent improvements on different GPUs

| Baseline Implementation | MinkowskiEngine 0.5.4 | SPConv 1.2.1 (FP32) | SPConv 1.2.1 (FP16) | TorchSparse |
|-------------------------|-----------------------|---------------------|---------------------|-------------|
| SK-MinkUNet (1.0x)      | 1.00                  | 1.06                | 1.00                | 1.00        |
| SK-MinkUNet (0.5x)      | 0.67                  | 0.65                | 0.65                | 0.67        |
| NS-MinkUNet (3f)        | 0.65                  | 0.50                | 0.50                | 0.65        |
| NS-MinkUNet (1f)        | 0.75                  | 0.50                | 0.50                | 0.75        |
| NS-CenterPoint (10f)    | 0.70                  | 0.85                | 0.85                | 0.70        |
| WM-CenterPoint (3f)     | 0.60                  | 0.69                | 0.69                | 0.60        |
| WM-CenterPoint (1f)     | 0.62                  | 0.69                | 0.69                | 0.62        |
| Geomean                 | 1.00                  | 1.00                | 1.00                | 1.00        |

---

**RTX 3090**

- SK-MinkUNet (1.0x)
- SK-MinkUNet (0.5x)
- NS-MinkUNet (3f)
- NS-MinkUNet (1f)
- NS-CenterPoint (10f)
- WM-CenterPoint (3f)
- WM-CenterPoint (1f)
- Geomean

---

**RTX 2080Ti**

- SK-MinkUNet (1.0x)
- SK-MinkUNet (0.5x)
- NS-MinkUNet (3f)
- NS-MinkUNet (1f)
- NS-CenterPoint (10f)
- WM-CenterPoint (3f)
- WM-CenterPoint (1f)
- Geomean

---

**RTX 1080Ti**

- SK-MinkUNet (1.0x)
- SK-MinkUNet (0.5x)
- NS-MinkUNet (3f)
- NS-MinkUNet (1f)
- NS-CenterPoint (10f)
- WM-CenterPoint (3f)
- WM-CenterPoint (1f)
- Geomean

---

SK: SemanticKITTI (ICCV’19), NS: nuScenes (CVPR’20), WM: Waymo (CVPR’20)
Conclusion

Sparse convolution is an emerging operator in point cloud processing.

We trade computation for regularity, optimizing matrix multiplication in sparse convolution via adaptive grouping.

We reduce the memory footprint of sparse convolution via fused and locality-aware memory access.

🔗 https://torchsparse.mit.edu
🔗 https://github.com/mit-han-lab/torchsparse
Conclusion

Sparse convolution is an emerging operator in point cloud processing. We trade computation for regularity, optimizing matrix multiplication in sparse convolution via adaptive grouping. We reduce the memory footprint of sparse convolution via fused and locality-aware memory access.

🔗 https://torchsparse.mit.edu
🔗 https://github.com/mit-han-lab/torchsparse
Related work

**Full-stack efficient 3D perception for auto-driving**

Lidar and 3D point clouds: **sparse**, **irregular**, and **large** memory footprint => hardware unfriendly
Camera and 2D images: **high resolution**, **multi-camera**, **real-time** => computationally hungry
Sensor fusion: multiple sensors, multiple tasks => even more computationally hungry

---

**Algorithm**

3D Light-weight Neural-net [PVCNN, NeurIPS’19 Spotlight]
3D Neural Architecture Search [SPVNAS, ECCV’20]
Multi-Task Muti Sensor Fusion [BEVFusion, arXiv’22]

**Software**

3D Inference Engine on GPU [TorchSparse, MLsys’22]

**Hardware**

3D Hardware Accelerator [PointAcc, MICRO’21]

---

Full-stack optimizations

Camera x 10
LiDAR x 5
Radar x 6

Total Computation: >200T FLOPs

Video credit: https://semantic-kitti.org
Sparse convolution is an emerging operator in point cloud processing. We trade computation for regularity, optimizing matrix multiplication in sparse convolution via adaptive grouping. We reduce the memory footprint of sparse convolution via fused and locality-aware memory access.

https://torchsparse.mit.edu

https://github.com/mit-han-lab/torchsparse

Thank you!