# TorchSparse: Efficient Point Cloud Inference Engine

**Haotian Tang** [*1] **Zhijian Liu** [*1] **Xiuyu Li** [*2] **Yujun Lin** [1] **Song Han** [1]

https://torchsparse.mit.edu

## Abstract

Deep learning on point clouds has received increased attention thanks to its wide applications in AR/VR and autonomous driving. These applications require low latency and high accuracy to provide real-time user experience and ensure user safety. Unlike conventional dense workloads, the sparse and irregular nature of point clouds poses severe challenges to running sparse CNNs efficiently on the general-purpose hardware. Furthermore, existing sparse acceleration techniques for 2D images do not translate to 3D point clouds. In this paper, we introduce TorchSparse, a high-performance point cloud inference engine that accelerates the sparse convolution computation on GPUs. TorchSparse directly optimizes the two bottlenecks of sparse convolution: **irregular computation** and **data movement**. It applies *adaptive matrix multiplication grouping* to trade computation for better regularity, achieving 1.4-1.5× speedup for matrix multiplication. It also optimizes the data movement by adopting *vectorized*, *quantized* and *fused locality-aware memory access*, reducing the memory movement cost by 2.7×. Evaluated on seven representative models across three benchmark datasets, TorchSparse achieves **1.6**× and **1.5**× measured end-to-end speedup over the state-of-the-art MinkowskiEngine and SpConv, respectively.

## 1 Introduction

3D point cloud becomes increasingly accessible over the past few years thanks to the widely available 3D sensors, such as LiDAR scanners (on the self-driving vehicles and, more recently, even the mobile phones) and depth cameras (on the AR/VR headsets). Compared with 2D RGB images, 3D point clouds provide much more accurate spatial/depth information and are usually more robust to different lighting conditions. Therefore, 3D point cloud processing becomes the key component of many real-world AI applications: *e.g.*, to understand the indoor scene layout for AR/VR, and to parse the driveable regions for autonomous driving.

A 3D point cloud is an *unordered* set of 3D points. Unlike 2D image pixels, this data representation is highly sparse and irregular. Researchers have explored to rasterize the 3D data into dense volumetric representation (Qi et al., 2016) or directly process it in the point cloud representation (Qi et al., 2017b; Li et al., 2018). However, both of them are not scalable to large indoor/outdoor scenes (Liu et al., 2019). Alternatively, researchers have also investigated to flatten 3D point clouds into dense 2D representations using spherical projection and bird's-eye view (BEV) projection. However, their accuracy is much lower due to the physical dimension distortion and height information loss.

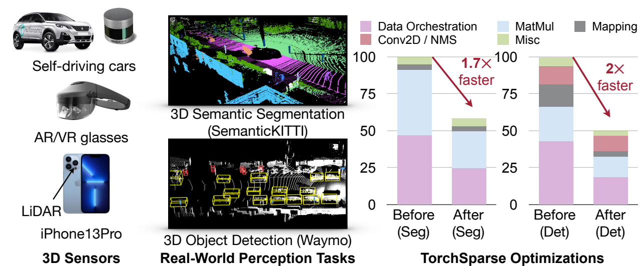



Figure 1: Widely available 3D sensors (left) have enabled more and more real-world AI applications to perceive the world using 3D point clouds (middle). However, processing 3D point cloud requires sparse computation that is not favored by general-purpose hardware. TorchSparse reduces the irregular computation and optimizes the data movement, achieving 1.7× to 2× measured speedup (right).

Recently, state-of-the-art 3D point cloud neural networks tend to rely largely or fully on sparse convolutions (Graham et al., 2018), making it an important workload for machine learning system: all top 5 segmentation submissions on SemanticKITTI (Behley et al., 2019) are based on SparseConv,

[*]Equal contribution [1]Massachusetts Institute of Technology [2]Cornell University. Correspondence to: Song Han <songhan@mit.edu>.

*Proceedings of the* $5^{th}$ *MLSys Conference*, Santa Clara, CA, USA, 2022. Copyright 2022 by the author(s).

9 of top 10 submissions on nuScenes (Caesar et al., 2020) and top 2 winning solutions on Waymo (Sun et al., 2020) have exploited SparseConv-based detectors (Yin et al., 2021; Ge et al., 2021). Given the wide applicability and dominating performance of SparseConv-based point cloud neural networks, it is crucial to provide efficient system support for sparse convolution on the general-purpose hardware.

Unlike conventional dense computation, sparse convolution is not supported by existing inference libraries (such as TensorRT and TVM), which is why most industrial solutions still prefer 2D projection-based models despite their lower accuracy. It is urgent to better support the sparse workload, which was not favored by modern high-parallelism hardware. On the one hand, the sparse nature of point clouds leads to irregular computation workloads: *i.e.*, different kernel offsets might correspond to drastically different numbers of matched input/output pairs. Hence, existing sparse inference engines (Yan et al., 2018; Choy et al., 2019) usually execute the matrix multiplication for each kernel offset separately, which cannot fully utilize the parallelism of modern GPUs. On the other hand, neighboring points do not lie contiguously in the sparse point cloud representation. Explicitly gathering input features and scattering output results can be very expensive, taking up to 50% of the total runtime. Due to the irregular computation workload and expensive data movement cost, SparseConv-based neural networks can hardly be run in real time: the latest sparse convolution library can only run MinkowskiNet at 8FPS on an NVIDIA GTX 1080Ti GPU, let alone other low-power edge devices.

In this paper, we introduce TorchSparse, a high-performance inference engine tailored for sparse point cloud computation. TorchSparse is optimized based upon two principles: (1) improving the computation regularity and (2) reducing the memory footprint. First, we propose the adaptive matrix multiplication grouping to batch the computation workloads from different kernel offsets together, trading #FLOPs for regularity. Then, we adopt quantization and vectorized memory transactions to reduce memory movement. Finally, we gather and scatter features in locality-aware memory access order to maximize the data reduce. Evaluated on seven models across three datasets, TorchSparse achieves **1.6**$\times$ and **1.5**$\times$ speedup over state-of-the-art MinkowskiEngine and SpConv, paving the way for deploying 3D point cloud neural networks in real-world applications.

## 2 BACKGROUND

The point cloud can be formulated as an unordered set of points paired with features: $\{(\boldsymbol{p}_j, \boldsymbol{x}_j)\}$, where $\boldsymbol{x}_j \in \mathbb{R}^C$ is a $C$-dimensional feature vector for point $\boldsymbol{p}_j \in \mathbb{Z}^D$ in a $D$-dimensional space. For a convolution of kernel size $K$, let $\boldsymbol{W} \in \mathbb{R}^{K^D \times C^{\text{in}} \times C^{\text{out}}}$ be its weights and $\boldsymbol{\Delta}^D(K)$ be its kernel offsets (*e.g.*, $\boldsymbol{\Delta}^2(5) = \{-2,-1,0,1,2\}^2$ and $\boldsymbol{\Delta}^3(3) = \{-1,0,1\}^3$). The weights $\boldsymbol{W}$ can be broken down into $K^D$ matrices of shape $C^{\text{in}} \times C^{\text{out}}$, denoted as $\boldsymbol{W_\delta}$ for $\boldsymbol{\delta} \in \boldsymbol{\Delta}^D(K)$. With these notations, the convolution with stride $s$ can be represented as

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

where $\boldsymbol{p}_j \in \boldsymbol{P}^{\text{in}}$, $\boldsymbol{q}_k \in \boldsymbol{P}^{\text{out}}$, and $1[\cdot]$ is the binary indicator.

For dense convolution (Figure 3a), each nonzero input is multiplied with all nonzero weights, leading to rapidly growing nonzeros ($\boldsymbol{P}^{\text{in}} \subset \boldsymbol{P}^{\text{out}}$). On the other hand, the computation of sparse convolution (Figure 3b) is determined by ***maps*** $\mathcal{M} = \{(\boldsymbol{p}_j, \boldsymbol{q}_k, \boldsymbol{W_\delta})\}$ in Equation 1 (also written as $\{(\boldsymbol{p}_j, \boldsymbol{q}_k, \boldsymbol{W}_n)\}$, where $n$ is the weight index) and keeps the sparsity pattern unchanged ($\boldsymbol{P}^{\text{in}} = \boldsymbol{P}^{\text{out}}$). It iterates over all maps and performs $\boldsymbol{x}_k^{\text{out}} = \boldsymbol{x}_k^{\text{out}} + \boldsymbol{x}_j^{\text{in}} \cdot \boldsymbol{W_\delta}$.

### 2.1 Mapping Operations

Mapping is a step to construct the input-output *maps* $\mathcal{M} = \{(\boldsymbol{p}_j, \boldsymbol{q}_k, \boldsymbol{W_\delta})\}$ for sparse convolution. Here, $j$ is the index of input point $\boldsymbol{p}$ in $\boldsymbol{P}^{\text{in}}$, $k$ is the index of output point $\boldsymbol{q}$ in $\boldsymbol{P}^{\text{out}}$, and $\boldsymbol{W_\delta}$ is the weight matrix for kernel offset $\boldsymbol{\delta}$. Generating maps typically requires two steps: calculating the output coordinates $\boldsymbol{P}^{\text{out}}$, and searching maps $\mathcal{M}$. These operations only take coordinates as input.

#### 2.1.1 *Output Coordinates Calculation*

When the convolution stride is 1, the output coordinates are exactly the same as the input coordinates, *i.e.*, $\boldsymbol{P}^{\text{out}} = \boldsymbol{P}^{\text{in}}$.

When the convolution stride is larger than 1, the nonzero input coordinates will first be dilated for each kernel offset (*i.e.*, $\boldsymbol{p}-\boldsymbol{\delta}$). After that, only these points on the strided grids within boundaries will become outputs $\boldsymbol{q}$, where $s{\cdot}\boldsymbol{q} = \boldsymbol{p}-\boldsymbol{\delta}$. Take the input coordinate $(3,5)$ as an example (with stride of 2). For offset $\boldsymbol{\delta} = (1,1)$, the output coordinate will be $((3,5)-(1,1))/2 = (1,2)$, while for offset $\boldsymbol{\delta} = (0,0)$, there is no valid output coordinate since $((3,5)-(0,0))$ is not a multiple of stride $s = 2$. We refer the readers to Appendix A for more details.

#### 2.1.2 *Map Search*

As in Algorithm 1, map search requires iterating over all possible input coordinates for each output coordinate. A map is generated only when the input is nonzero.

To efficiently examine whether the possible input $\boldsymbol{q}_j + \boldsymbol{\delta}$ is nonzero, a common implementation is to record the coordinates of nonzero inputs with a hash table. The key-value pairs are (key=input coordinates, value=input index), *i.e.*, $(\texttt{key} = \boldsymbol{p}_j, \texttt{value} = j)$. The hash function can simply be flattening the coordinate of each dimension into an integer.

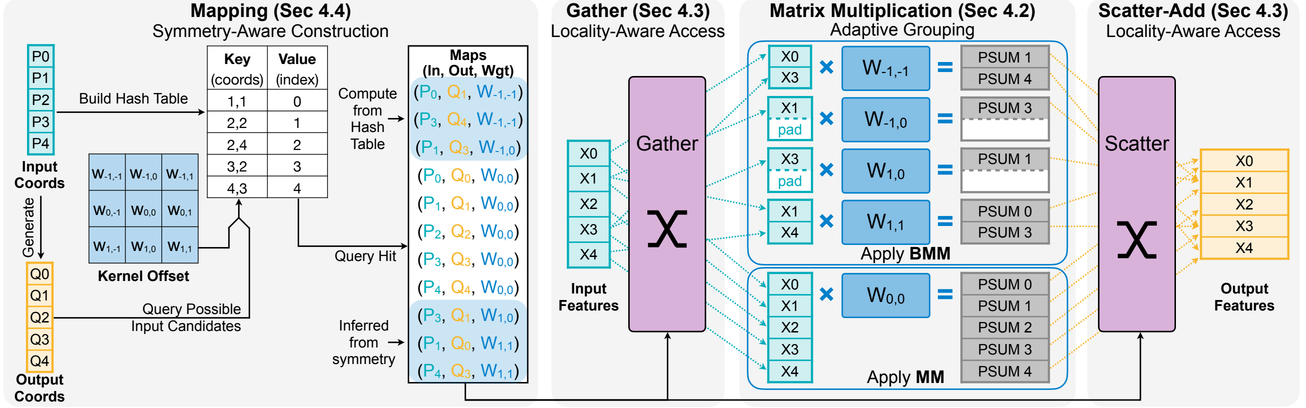



Figure 2: TorchSparse aims at accelerating *Sparse Convolution*, which consists of four stages: mapping, gather, matmul and scatter-accumulate. Our optimization follows two principles: ❶ improve the regularity of sparse workload ❷ reduce the memory footprint. To achieve that, TorchSparse exploits adaptively batched matmul (Principle ❶, Section 4.2); quantized, vectorized, locality-aware scatter/gather (Principle ❷, Section 4.3); and mapping kernel fusion (Principle ❷, Section 4.4).

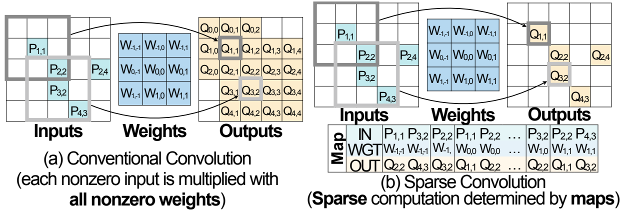



Figure 3: Sparse convolution (b) does *not* multiply each nonzero input with all nonzero weights as conventional convolution (a) does.

### 2.2 Data Orchestration and Matrix Multiplication

After maps are generated, sparse convolution will multiply the input feature vector $\boldsymbol{x}_j^{\text{in}}$ with corresponding weight matrix $\boldsymbol{W}_{\boldsymbol{\delta}}$ and accumulate to the corresponding output feature vector $\boldsymbol{x}_k^{\text{out}}$, following the map $\{(\boldsymbol{p}_j, \boldsymbol{q}_k, \boldsymbol{W}_{\boldsymbol{\delta}})\}$.

The utilization of matrix-vector multiplication is rather low on GPU. Therefore, most existing implementations follow the gather-matmul-scatter computation flow in Algorithm 2. First, all input feature vectors associated with the same weight matrix are gathered and concatenated into a contiguous matrix. Then, matrix-matrix multiplication between feature matrix and weight matrix is conducted to obtain the partial sums. Finally, these partial sums are scattered and accumulated to the corresponding output feature vectors.

### 2.3 Difference from Other Tasks

***vs*. conventional convolution with sparsity.** The sparsity in conventional convolution comes from the ReLU activation function or weight pruning. Since there is no hard constraint on the output sparsity pattern, each nonzero input is multiplied with every nonzero weight, so the nonzeros will

**Algorithm 1** Map Search

**Input:** input coordinates $\boldsymbol{P}^{\text{in}}$, output coordinates $\boldsymbol{P}^{\text{out}}$
kernel size $K$, stride $s$
**Output:** maps $\mathcal{M}$

$N \leftarrow \boldsymbol{\Delta}^D(K)$.size
$\mathcal{M} \leftarrow \{\emptyset\} \times N$
**for** $k, \boldsymbol{q}_k$ **in** enumerate($\boldsymbol{P}^{\text{out}}$) **do**
  # Traverse the neighbors of an *output* point.
  **for** $n, \boldsymbol{\delta}$ **in** enumerate($\boldsymbol{\Delta}^D(K)$) **do**
    # Calculate *input* coordinates.
    $\boldsymbol{r} \leftarrow s \cdot \boldsymbol{q}_k + \boldsymbol{\delta}$
    # Add new map if input exists.
    **if** $\boldsymbol{P}^{\text{in}}$.contain($\boldsymbol{r}$) **then**
      $j \leftarrow \boldsymbol{P}^{\text{in}}$.getIndex($\boldsymbol{r}$)
      $\mathcal{M}[\boldsymbol{W}_{\boldsymbol{\delta}}] \leftarrow \mathcal{M}[\boldsymbol{W}_{\boldsymbol{\delta}}] \cup \{(\boldsymbol{p}_j, \boldsymbol{q}_k, \boldsymbol{W}_{\boldsymbol{\delta}})\}$
    **end if**
  **end for**
**end for**

**Algorithm 2** Gather-MatMul-Scatter

**Input:** input features $\boldsymbol{X}^{\text{in}}$, weights $\boldsymbol{W}$, maps $\mathcal{M}$
**Output:** output features $\boldsymbol{X}^{\text{out}}$

$\boldsymbol{X}^{\text{out}} \leftarrow \boldsymbol{0}$
# *Separately* perform gather-matmul-scatter for each weight.
**for** $\boldsymbol{\delta}$ **in** $\boldsymbol{\Delta}^D(K)$ **do**
  $\boldsymbol{F} \leftarrow \emptyset$
  # Gather features for $w_n$.
  **for** $m, (\boldsymbol{p}_j, \boldsymbol{q}_k, \boldsymbol{W}_{\boldsymbol{\delta}})$ **in** enumerate($\mathcal{M}[\boldsymbol{W}_{\boldsymbol{\delta}}]$) **do**
    $\boldsymbol{F}[m] \leftarrow \boldsymbol{X}^{\text{in}}[j]$
  **end for**
  # Matrix-matrix multiplication.
  $\boldsymbol{F} \leftarrow \boldsymbol{F} \cdot \boldsymbol{W}_{\boldsymbol{\delta}}$
  # Scatter partial sums to $\boldsymbol{X}^{\text{out}}$.
  **for** $m, (\boldsymbol{p}_j, \boldsymbol{q}_k, \boldsymbol{W}_{\boldsymbol{\delta}})$ **in** enumerate($\mathcal{M}[\boldsymbol{W}_{\boldsymbol{\delta}}]$) **do**
    $\boldsymbol{X}^{\text{out}}[\boldsymbol{q}_k] \leftarrow \boldsymbol{X}^{\text{out}}[\boldsymbol{q}_k] + \boldsymbol{F}[m]$
  **end for**
**end for**

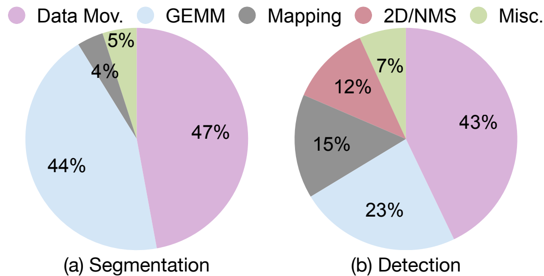



Figure 4: Data movement and GEMM constitute a significant proportion of the runtime of sparse CNNs.

dilate during the inference, *i.e.*, $\boldsymbol{P}^{\text{in}} \subset \boldsymbol{P}^{\text{out}}$ (see Figure 3a). The existing sparse computation libraries leverage such computation pattern by travelling all nonzero inputs with all nonzero weights to accelerate the conventional convolution. On the contrary, sparse convolution requires $\boldsymbol{P}^{\text{in}} = \boldsymbol{P}^{\text{out}}$, and thus the relationship among inputs, weights and outputs requires to be explicitly searched with mapping operations, which makes it a hassle for previous sparse libraries.

***vs*. graph convolution.** In graph convolution, the relationship between inputs and outputs are provided in the adjacency matrix which stays constant across layers. Contrarily, sparse convolution has to search maps for **every** downsampling block. Furthermore, graph convolution shares the same weight matrix for different neighbors, *i.e.* all $\boldsymbol{W}_\delta$ are the same. Hence, graph convolution only needs either one gather or one scatter of features: 1) first gather input features associated with the same output vertex, and then multiply them with shared weights and reduce to the output feature vector; or 2) first multiply all input features with shared weights, and then scatter-accumulate the partial sums to the corresponding output feature vector. However, sparse convolution uses different weight matrices for different kernel offset $\boldsymbol{\delta}$ and thus needs both gather and scatter during the computation. Consequently, existing SpMV/SpDMM systems for graph convolution acceleration (Wang et al., 2019a; Hu et al., 2020) are not applicable to sparse convolution.

## 3 ANALYSIS

We systematically profile the runtime of different components in two representative sparse CNNs: one for segmentation (Figure 4a) and one for detection (Figure 4b). Based on observations in Figure 4, we summarize two principles for sparse convolution optimization which lays the foundation for our system design in Section 4.

**Principle I. Improve Regularity in Computation** Matrix multiplication is the core computation in sparse convolution and takes up a large proportion of total execution time (20%-50%). Algorithm 2 decouples the matrix multiplication computation from data movement so that we can use well-optimized libraries (such as cuDNN) to calculate $\boldsymbol{X}^{\text{out}} \leftarrow \boldsymbol{X}^{\text{in}} \cdot \boldsymbol{W}_\delta$. However, the computation workloads are very *non-uniform* due to the irregular nature of point clouds (detailed in Figure 12, where map sizes for different weights can differ by an order of magnitude, and most map sizes are *small*). As a result, the matrix multiplication in MinkUNet (0.5$\times$ width) runs at 8.1 TFLOP/s on RTX 2080 Ti with FP16 quantization, achieving only 30% device utilization. Therefore, *improving the regularity* of matrix multiplication will potentially be helpful: we boost the utilization to 44.2% after optimization (detailed in Table 2).

**Principle II. Reduce Memory Footprint** Data movement is the largest bottleneck in sparse CNNs, which takes up 40%-50% of total runtime on average. This is because scatter-gather operations are bottlenecked by GPU memory bandwidth (*limited*) rather than computation resources (*abundant*). Worse still, the dataflow in Algorithm 2 completely separates scatter-gather operations for different kernel offsets. This further ruins the possibility of any reuse in the data movement, which will be detailed in Figure 9. It is also noteworthy that the large mapping latency in the CenterPoint detector (Figure 4b) also stems from memory overhead: hashmap construction and output coordinate calculation both require multiple DRAM accesses. Thus, *reducing memory footprint* is at the heart of data movement and mapping optimization.

## 4 SYSTEM DESIGN AND OPTIMIZATION

This section unfolds our TorchSparse as follows: Section 4.1 provides an overview and API design for TorchSparse, Section 4.2 introduces improvements on matrix multiplication operations, Section 4.3 elaborates the optimizations for data movement operations (scatter/gather), and Section 4.4 analyzes the opportunities to speed up mapping operations.

### 4.1 System Overview

Figure 5 provides an overview of our TorchSparse. At the top level, users define their sparse CNNs using TorchSparse APIs, which have minimal differences with native PyTorch APIs. Also, TorchSparse does not require users to add additional fields such as `indice_key` and `spatial_shape` in SpConv (Yan et al., 2018), and `coordinate_manager` in MinkowskiEngine (Choy et al., 2019) when defining modules and tensors. TorchSparse converts the high-level modules to primitive operations: *e.g.*, `Conv3d` is decomposed to output construction, mapping operations and gather-matmul-scatter. For each part, Python APIs interact with backend CUDA implementations via `pybind`. Note that TorchSparse also provides support for CPU inference and multi-GPU training, but this paper will focus on the GPU inference.

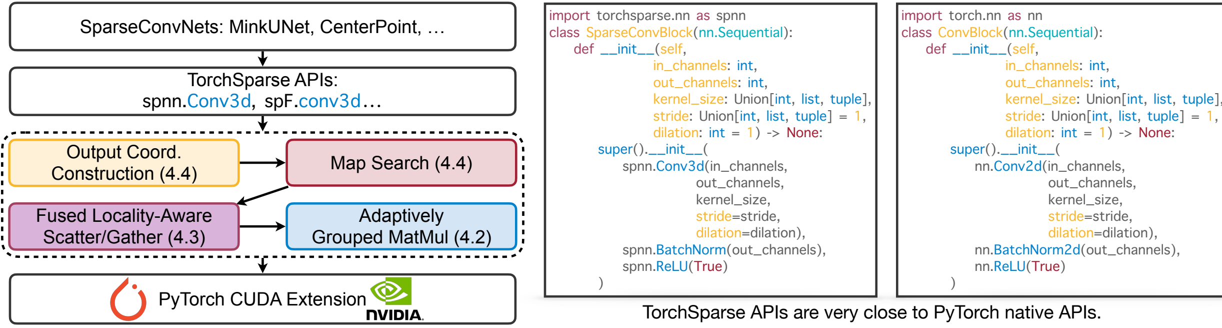



Figure 5: System overview for TorchSparse: our TorchSparse provides handy Python APIs similar to PyTorch and applies low-level optimizations to data movement, matrix multiplication and mapping operations in sparse convolution.

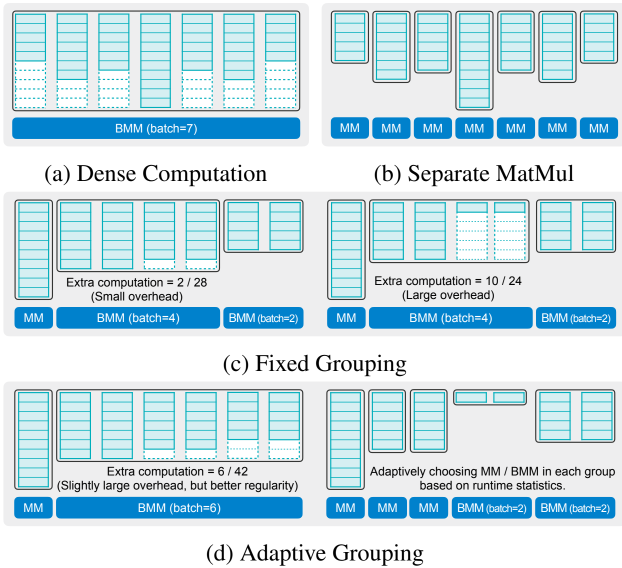



Figure 6: Different matrix multiplication grouping strategies: (a) dense computation suffers from large FLOPs overhead; (b) separate matrix multiplication suffers from low device utilization and excessive kernel calls; (c) fixed grouping trades FLOPs for regularity; (d) adaptive grouping searches for the best balance point.

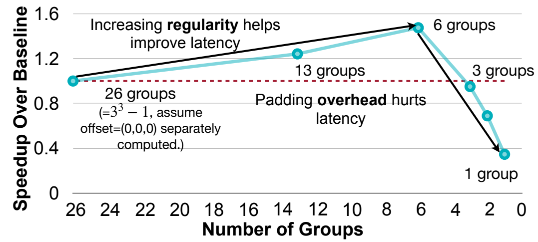



Figure 7: Trading FLOPs for computation regularity via batched matrix multiplication brings 1.5× speedup.

## 4.2 Matrix Multiplication Optimization

Matrix multiplication is the core computation in sparse convolution. Due to the irregularity of point clouds, existing implementations rely on cuDNN to perform many small matrix multiplications on different weights (Figure 6b), which usually do not saturate the utilization of GPUs. In order to increase the utilization, we propose to trade computation for regularity (Principle I) by grouping matrix multiplication for different weights together. We find it helpful to introduce redundant computation but group more computation in a single kernel. In Figure 12, we collect the real workload for MinkUNet (Choy et al., 2019) on SemanticKITTI (Behley et al., 2019) and analyze the efficiency of matrix multiplication in the first sparse convolution layer with respect to the group size. It turns out that batched matrix multiplication can be significantly faster than sequentially performing the computation along the batch dimension, thanks to the better regularity. This motivates us to explore the opportunity of *grouping* in the matrix multiplication computation.

### 4.2.1 *Symmetric Grouping*

With sparse workloads, the map sizes for different weights within one sparse convolution layer are usually *different*. Fortunately, for sparse convolutions with odd kernel size and stride of 1, the maps corresponding to kernel offset $(a, b, c)$ will always have the same size as the maps corresponding to the *symmetric* kernel offset $(-a, -b, -c)$. For a map entry $(\boldsymbol{p}_j, \boldsymbol{q}_k, \boldsymbol{W}_{a,b,c})$, we have $\boldsymbol{q}_k = \boldsymbol{p}_j + (a, b, c)$. Then, $\boldsymbol{p}_j = \boldsymbol{q}_k + (-a, -b, -c)$, which implies that $(\boldsymbol{q}_k, \boldsymbol{p}_j, \boldsymbol{W}_{-a,-b,-c})$ is also a valid map entry. As such, we can establish an one-to-one correspondence between maps for weights $\pm(a, b, c)$. Therefore, we are able to group the workload for symmetric kernel offsets together and naturally have a batch size of 2. Note that the workload corresponding to the kernel offset $(0, 0, 0)$ is processed separately since it does not require any explicit data movement. From Figure 7, the symmetric grouping (13 groups) can already be up to 1.2× faster than the separate matrix multiplication.

#### 4.2.2 *Fixed Grouping*

Though symmetric grouping works well for sparse convolutions with the stride of 1, it falls short in generalizing to downsampling layers. Also, it cannot push the batch size to $> 2$, which means that we still have a large gap towards the best GPU utilization in Figure 7. Nevertheless, we find that clear pattern exists in the map size statistics (Figure 12): for submanifold layers, the maps corresponding to $\boldsymbol{W}_0$ to $\boldsymbol{W}_3$ tend to have similar sizes and the rest of the weights other than the middle one have similar sizes; for downsampling layers, the maps for all offsets have similar sizes. Consequently, we can batch the computation into three groups accordingly. Within each group, we pad all features to the maximum size (Figure 6c). Fixed grouping generally works well when all features within the same group have similar sizes (Figure 6c left), and this usually happens in downsampling layers. For submanifold layers (Figure 6c right), the padding overhead can sometimes be large despite the better regularity, resulting in wasted computation.

#### 4.2.3 *Adaptive Grouping*

The major drawback of fixed grouping is that it does *not* adapt to individual samples. This can be problematic since workload size distributions can vary greatly across different datasets (Figure 12). It is also very labor-intensive to design different grouping strategies for different *layers*, different *networks* on a diverse set of *datasets* and *hardware*. To this end, we design an adaptive grouping algorithm (Figure 6d) that *automatically* determines the *input-adaptive* grouping strategy for a given layer on arbitrary workload.

The adaptive grouping algorithm builds upon two auto-tuned parameters $\epsilon$ and $S$, where $\epsilon$ indicates our tolerance of redundant computation, and $S$ is the workload threshold. Given $\epsilon$, we scan over sizes of all maps in the current workload for $\boldsymbol{W}_0$ to $\boldsymbol{W}_{K-1}$ (where $K$ is the kernel volume) and dynamically maintain two pointers indicating the start and end of the current group. We initiate a new group whenever the redundant computation ratio $(1 - \frac{\text{Theoretical FLOPs}}{\text{Actual FLOPs}})$ exceeds $\epsilon$. Then, given $S$, we inspect the maximum workload size within each group. Each group performs `bmm` if the workload size is smaller than $S$ and performs `mm` otherwise. This is because `bmm` can improve device utilization for small workloads but has little benefit for large workloads. We refer the readers to Appendix B for more details of this algorithm. Note that even if $\epsilon$ and $S$ are fixed, the generated strategy itself is still *input-adaptive*. Since different input point clouds have different map sizes, even the same $\epsilon$ can potentially generate different group partition strategies for different samples. The $(\epsilon, S)$ parameter space is simple but diverse enough to cover dense computation ($\epsilon = 1$; $S = +\infty$), separate computation ($S = 0$) as well as symmetric grouping ($\epsilon = 0$; $S = +\infty$) as its special cases.

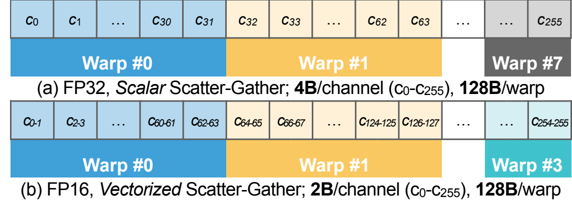



Figure 8: TorchSparse applies *vectorized* and *quantized* scatter-gather to greatly reduce the data movement latency.

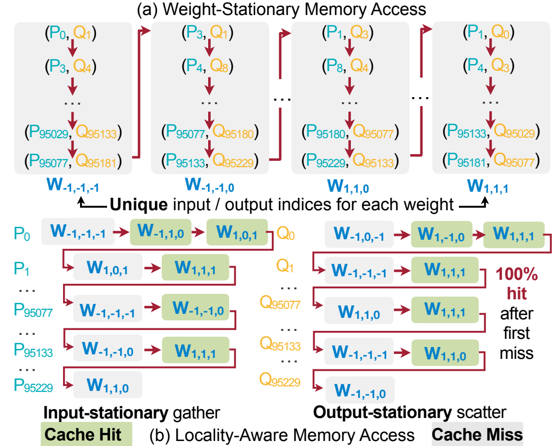



Figure 9: TorchSparse proposes cache-friendly *locality-aware* and memory access pattern. In contrary, baseline implementation (a) cannot exploit cache reuse due to uniqueness in input/output indices for each weight.

For a given sparse CNN, we determine $(\epsilon, S)$ for each layer on the target dataset and hardware platform via exhaustive grid search on a small subset (usually 100 samples) of the training set. We formalize this process in Appendix B. The search is inference-only. It explores a space of around 1,000 configurations and requires less than 10 minutes of search time on a desktop GPU. The strategy derived on the small subset can be directly applied and does not require any parameter optimization during the inference time.

### 4.3 Data Movement Optimization

From Section 3, data movement usually takes up 40-50% of the total runtime. Thus, optimizing data movement will be of high priority as well (Principle II). Intuitively, it is most effective to reduce data movement cost by reducing the total amount of DRAM access and exploiting the data reuse.

#### 4.3.1 *Quantized and Vectorized Memory Access*

FP16 quantization brings $2\times$ theoretical DRAM access saving compared with the FP32 baseline. However, as in Figure 8, this reduction cannot be translated into real speedup without vectorized scatter/gather.

NVIDIA GPUs group memory access requests into *transactions*, whose largest size is 128 bytes. Considering the typical memory access pattern in scatter/gather, where a warp (32 threads) issues contiguous FP32 (4 bytes) memory access instructions simultaneously, the 128-byte transaction is fully utilized. However, when each thread in the warp issues an FP16 memory request, the memory transaction has only 64/128=50% utilization, and the total number of memory transactions are essentially unchanged. As a result, we observe far smaller speedup (1.3×) compared to the theoretical value (2×) on scatter/gather if *scalar* scatter/gather (Figure 8a) is performed.

Contrarily, *vectorized* scatter-gather Figure 8b doubles the workload of each thread, making the total work of each warp still 128 bytes, equivalent to a full FP32 memory transaction. Meanwhile, the total number of memory transactions is *halved* while the work for each memory transaction is *unchanged*, and we observe **1.9**× speedup over FP32 data movement on various GPU platforms. This closely aligns with the theoretical reduction in DRAM access.

Further quantizing the features to INT8 offers diminishing return, as the multi-way reduction in the scatter operation requires more than 8-bit for the final result. In this case, all scatter operations are still in 16 bits since CUDA requires aligned memory access. Thus, scattering (which takes 60% of the data movement time) cannot not accelerated with the INT8 quantization, leading to limited overall speedup.

#### 4.3.2 *Fused and Locality-Aware Memory Access*

Despite the limitation of aggressive feature quantization, it is still possible to achieve faster scatter/gather by exploiting locality. Intuitively, for a sparse convolution layer, the total amount of gather read and scatter write is $N_1 = |\mathcal{M}|(C_{\text{in}}+C_{\text{out}})$, where $\mathcal{M}$ is the map for this layer (defined in Section 2.1), and $C_{\text{in}}$ and $C_{\text{out}}$ correspond to input and output channel numbers. However, the total feature size of this layer is $N_2 = N_{\text{in}}C_{\text{in}} + N_{\text{out}}C_{\text{out}}$. Empirically, the feature of each point is repetitively accessed for at least 4 times ($N_1 \geq 4N_2$). Based on this, we can ideally have **1.6**× more DRAM access saving for scatter/gather (the amount of gather write and scatter read is also $N_1$ and cannot be saved).

As shown in Algorithm 2 and Figure 9a, the current implementation completely separates gather/scatter for different weights. When we perform *gather* operation for $\boldsymbol{W}_{k+1}$, the GPU cache is filled with *scatter* buffer features for $\boldsymbol{W}_k$ as long as the GPU cache size is much smaller than $N_1$ (typically $>$ 40MB, much larger than the 5.5MB L2 cache of NVIDIA RTX 2080 Ti). Intuitively, for gather operation on $\boldsymbol{W}_{k+1}$, we hope that the cache is filled with gather buffer features from $\boldsymbol{W}_k$. This suggests us to first fuse *all* gather operations before matrix multiplication, and fuse *all* scatter operations afterwards. As such, the GPU cache will always hold data from the same type of buffer.

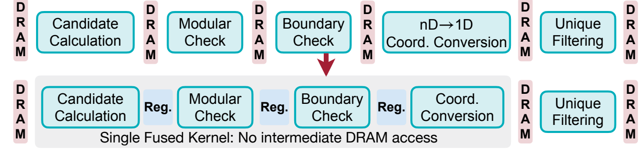



Figure 10: TorchSparse reduces mapping DRAM access and improves mapping latency via kernel fusion.

Moreover, the memory access order matters. In the weight-stationary order (Figure 9a), all map entries for weight $\boldsymbol{W}_k$ are *unique*, so there is *no* chance of feature reuse, and each gather/scatter leads to a cache miss. As in Figure 9b, we instead take a *locality-aware* memory access order. We gather the input features in the *input-stationary* order and scatter the partial sums in the *output-stationary* order.

Without loss of generality, we will focus on the implementation of input-stationary gather. We first maintain a neighbor set $\mathcal{N}_j$ for each input point $\boldsymbol{p}_j$: *i.e.*, for the i$^{\text{th}}$ map entry $(\boldsymbol{p}_j, \boldsymbol{q}_k, \boldsymbol{W}_n)$, we insert $(\boldsymbol{W}_n, i)$ into $\mathcal{N}_j$. Then, we iterate over every input point $\boldsymbol{p}_j$, fetch its feature vector $\boldsymbol{X}^{\text{in}}_j$ into the register, and write it to the corresponding DRAM location $\sum_{k=0}^{n-1} |\mathcal{M}[\boldsymbol{W}_k]| + i$ for each $(\boldsymbol{W}_n, i) \in \mathcal{N}_j$. Here, $\mathcal{M}[\boldsymbol{W}_k]$ is the map for weight $\boldsymbol{W}_k$. Note that each $\boldsymbol{X}^{\text{in}}_j$ is read from DRAM only once and held in the register. Hence, this algorithm achieves the optimal reuse for gather. Similar technique can be applied to scatter, where we read neighbors' partial sums for each output point from DRAM, perform reduction in the register, and write the result back only once. This optimization alone leads to **1.3-1.4**× speedup in data movement on real-world point cloud datasets.

### 4.4 Mapping Optimization

From Figure 4, mapping operations in our baseline implementation take up a significant amount of time (15%) in detectors on the Waymo (Sun et al., 2020) dataset. It is important to reduce the mapping overhead in sparse CNNs.

We first choose the map search strategy for each layer from [`grid`, `hashmap`] in a similar manner to the adaptive grouping. Here, `grid` corresponds to a naive collision-free grid-based hashmap: it takes larger memory space, but hashmap construction/query requires exactly one DRAM access per-entry, which is much smaller than conventional hashmaps. We then perform kernel fusion (Figure 10) on output coordinates computation for downsampling. The downsample operation applies a sliding window around each point. It ❶ calculates candidate activated points with `broadcast_add`, ❷ performs modular check, ❸ performs boundary check and generates a mask on whether each point is kept, ❹ converts the remaining candidate point coordinates to 1D values,

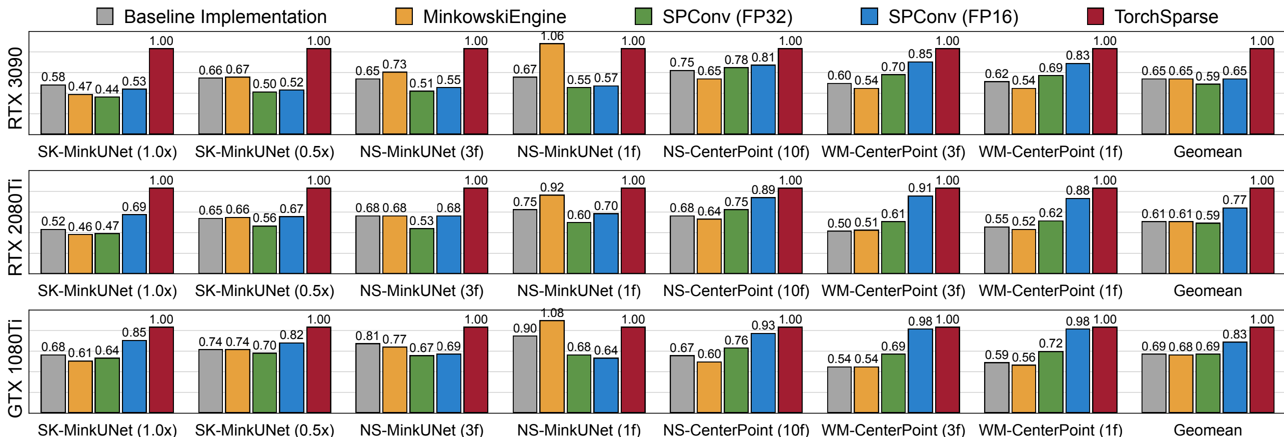
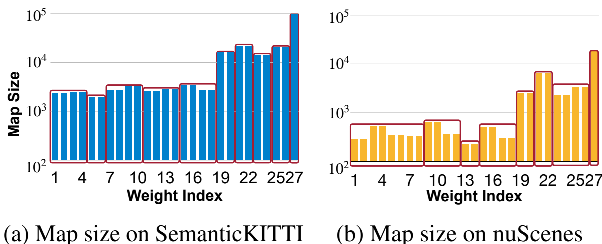



Figure 11: TorchSparse consistently outperforms state-of-the-art inference engines in both detection and segmentation benchmarks and achieves up to 1.5-1.6× geomean speedup, 2.3× single model speedup over MinkowskiEngine and SpConv.

and ❺ performs `unique` operation to keep final output coordinates (detailed in Appendix A). There are DRAM accesses between every two of the five stages, making downsampling kernels memory-bounded. We therefore fuse stages ❶ to ❹ into a single kernel and use registers to store intermediate results, which eliminates all intermediate DRAM write. For the fused kernel, we further perform control logic simplification, full loop unrolling and utilize the symmetry of submanifold maps. Overall, the mapping operations are accelerated by **4.6**× on detection tasks with our optimizations.




(a) Map size on SemanticKITTI (b) Map size on nuScenes

Figure 12: Grouping strategy on different datasets. Maps on nuScenes are much smaller than on SemanticKITTI for MinkUNet. Thus, to fully utilize GPU, the grouping strategy is more aggressive on nuScenes (8 groups *vs.* 10 groups).

## 5 Evaluation

### 5.1 Setup

TorchSparse is implemented in CUDA and provides easy-to-use PyTorch-like interfaces (described in Section 4.1). We build TorchSparse based on PyTorch 1.9.1 with CUDA 10.2/11.1 and cuDNN 7.6.5. Our system is evaluated against a baseline FP32 design without optimizations in Section 4 and the latest versions of two state-of-the-art sparse convolution libraries MinkowskiEngine v0.5.4 (Choy et al., 2019) and SpConv v1.2.1 (Yan et al., 2018) on three generations of NVIDIA GPUs: GTX 1080Ti, RTX 2080Ti and RTX 3090. Necessary changes are made to MinkowskiEngine to correctly support downsample operations in detectors and to SpConv to avoid OOM in large-scale scenes.

All systems are evaluated on seven top-performing sparse CNNs on large-scale datasets: MinkUNet (Choy et al., 2019) (0.5×/1× width) on SemanticKITTI (Behley et al., 2019), MinkUNet (1/3 frames) on nuScenes-LiDARSeg (Caesar et al., 2020), CenterPoint (Yin et al., 2021) (10 frames) on nuScenes detection and CenterPoint (1/3 frames) on Waymo Open Dataset (Sun et al., 2020). We report the normalized FPS for all systems (with TorchSparse to be 1).

### 5.2 Evaluation Results

Our TorchSparse achieves the best performance compared with the baseline design, MinkowskiEngine and SpConv.

From Figure 11, TorchSparse achieves up to 2.16× speedup on segmentation models and 1.6-2× speedup on detection models over MinkowskiEngine on RTX 3090. We achieve a smaller speedup for the 1-frame MinkUNet on nuScenes-LiDARSeg because MinkowskiEngine applies specialized optimizations to small models by using the *fetch-on-demand* dataflow (Lin et al., 2021) instead of the gather-matmul-scatter dataflow.

TorchSparse also demonstrates a 1.2× faster inference speed compared with the FP16 version of SpConv for detectors on RTX3090 thanks to our fused and locality-aware access pattern and almost perfect speedup from vectorized data movement. Note that we report *end-to-end* speedup in Figure 11. However, 10% of total total runtime in CenterPoint (Yin et al., 2021) is not related to point cloud computa-

tion (image convolution and non-maximum suppression, as in Figure 4). Therefore, our speedup ratio on sparse convolution is 10% more for CenterPoint. The performance gain over SpConv (FP16) is even larger on segmentation models on various hardware platforms thanks to the effectiveness of adaptively batched matrix multiplication, which will be discussed in Section 6.1. GPUs are usually more under-utilized for segmentation models as they usually have smaller workload compared with detectors, making it necessary to apply batching strategies to improve the device utilization.

TorchSparse achieves consistent speedup over other systems on GTX 1080Ti, which has *no* FP16 tensor cores. Compared with the baseline design, our TorchSparse still achieves a 1.5× speedup, only 11% less than the speedup we achieved on RTX 2080Ti with tensor cores. This validates that the native tensor-core speedup only constitutes a very minor proportion of our performance gain.

Our TorchSparse runs MinkUNet (1.0× on SemanticKITTI) at 36, 26 and 13 FPS on RTX 3090, RTX 2080Ti, GTX 1080Ti, respectively, all satisfying the real-time requirement ($\geq$ 10 FPS). For the 3-frame model on nuScenes-LiDARSeg, TorchSparse achieves 45, 40 and 25 FPS throughput on the three devices, at least 2× faster than the LiDAR frequency. Even for the heaviest 3-frame CenterPoint model on Waymo, our TorchSparse is still able to achieve the real-time inference on GTX 1080Ti. As such, our system paves the way for real-time LiDAR perception on self-driving cars.

## 6 Ablation Study

### 6.1 Matrix Multiplication Optimizations

We first examine the performance of different grouping strategies on SemanticKITTI with MinkUNet (0.5×) and on nuScenes with MinkUNet (3 frames). From Figure 6, our adaptive grouping strategy outperforms all handcrafted, fixed strategy and achieves 1.4-1.5× over no grouping baseline. Table 2 also suggests that manually-designed strategy cannot generalize to all datasets: fixed 3-batch grouping achieves large speedup (1.5×) on nuScenes, but is 13% slower than the separate computation baseline on SemanticKITTI. Note that although this strategy has the best device utilization (largest TFLOP/s) on nuScenes, it does not bring greater latency reduction than adaptive grouping due to much more extra computation, indicating the importance of $\epsilon$ in our adaptive grouping algorithm. We also show the effectiveness of grouping strategy specialization for different datasets, model and hardware in Table 1. In Table 1a, we found that the same model (1-frame MinkUNet) on the same hardware platform benefits more from the dataset-specialized strategy. This is because map size distributions (which decides the workload of matrix multiplication) significantly differ between SemanticKITTI and nuScenes, as shown in Figure 12. The maps on nuScenes are much smaller than those on SemanticKITTI. As a result, if we directly transfer SemanticKITTI strategy to nuScenes, the groups will not be large enough to fully utilize hardware resources. On the other hand, if the nuScenes strategy is transferred to SemanticKITTI, the efficiency will be bottlenecked by computation overhead. We notice similar effect for model and hardware specialization in Table 1b and Table 1c, where specialized strategies always outperform the transferred ones.

| Specialization for Different Datasets | | Optimized for: SemanticKITTI | Optimized for: nuScenes |
|---|---|---|---|
| Execute on | SemanticKITTI | **10.11** | 10.87 |
| | nuScenes | 5.30 | **4.67** |

(a) Specialization for Datasets (MinkUNet, 2080Ti)

| Specialization for Different Models | | Optimized for: MinkUNet (1.0×) | Optimized for: MinkUNet (0.5×) |
|---|---|---|---|
| Execute on | MinkUNet (1.0×) | **10.11** | 10.70 |
| | MinkUNet (0.5×) | 5.37 | **4.72** |

(b) Specialization for Model (SemanticKITTI, 2080Ti)

| Specialization for Different Hardware | | Optimized for: RTX2080Ti | Optimized for: GTX1080Ti |
|---|---|---|---|
| Execute on | RTX2080Ti | **4.67** | 4.80 |
| | GTX1080Ti | 14.95 | **14.01** |

(c) Specialization for Hardware (nuScenes, MinkUNet)

Table 1: Specializing adaptive batching strategies for different datasets, models and hardware platforms helps improve efficiency (TFLOP/s) by up to 13.5%.

| Grouping Method | MatMul speedup (SK) | MatMul Speedup (NS) |
|---|---|---|
| Separate | 8.1 TFLOP/s (1.00×) | 10.4 TFLOP/s (1.00×) |
| Symmetric | 8.2 TFLOP/s (1.02×) | 14.6 TFLOP/s (1.39×) |
| Fixed | 8.7 TFLOP/s (0.87×) | **21.1 TFLOP/s** (1.50×) |
| Adaptive | **11.9 TFLOP/s (1.39×)** | 16.9 TFLOP/s (**1.54**×) |

Table 2: Ablation analysis on matrix multiplication: adaptive batching consistently outperforms all other strategies in latency and brings about 1.4×-1.5× speedup for matmul (SK=SemanticKITTI, NS=nuScenes). As we trade FLOPs for regularity, TFLOP/s and speedup are non-proportional.

### 6.2 Data Movement Optimizations

We then perform ablation analysis on MinkUNet (Choy et al., 2019) (1.0×) on the SemanticKITTI dataset (Behley et al., 2019). As in Table 3, naively quantizing features to

| FP16 | Vec. | Fused | Loc.-Aware | Speedup (G) | Speedup (S) | Speedup (SG) |
|---|---|---|---|---|---|---|
| ✗ | ✗ | ✗ | ✗ | 1.00 | 1.00 | 1.00 |
| ✓ | ✗ | ✗ | ✗ | 1.17 | 1.48 | 1.32 |
| ✓ | ✓ | ✗ | ✗ | 1.91 | 1.95 | 1.93 |
| ✓ | ✓ | ✓ | ✗ | 1.91 | 2.12 | 2.02 |
| ✓ | ✓ | ✓ | ✓ | **2.86** | **2.61** | **2.72** |

Table 3: Speedup breakdown of different optimizations to reduce data movement. Feature quantization, vectorized memory access, and fused and locality-aware access bring 1.3×, 1.5× and 1.4× speedup, respectively. Here, *G* and *S* denote gather and scatter.

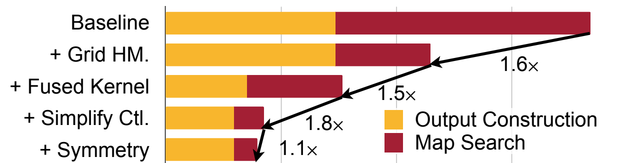



Figure 13: Speedup breakdown of mapping optimizations. Grid-based hashmap, fused kernel, simplified control logic and symmetry bring 1.6×, 1.5×, 1.8× and 1.1× measured speedup, respectively.

16-bit will not provide significant speedup for scatter/gather: especially for gather, the speedup ratio is only 1.17×, far less than the theoretical value (2×). Instead, quantized and *vectorized* scatter/gather improves the latency of scatter-gather by 1.93×, which closely matches the DRAM access reduction and verifies our analysis in Section 4.3 on memory transactions. We further observe that fusing gather/scatter itself will *not* provide substantial speedup, as the *weight-stationary* access pattern cannot provide good cache locality due to the uniqueness of maps for each weight. However, when combined with *locality-aware* access, we achieve **2.86**× speedup on gathering, **2.61**× speedup on scattering and **2.72**× overall speedup against FP32. This demonstrates the fact that all techniques in Section 4.3 are crucial in improving the efficiency of data movement.

### 6.3 Mapping Optimizations

We finally present analysis on optimizing mapping operations in 3-frame CenterPoint (Yin et al., 2021) detector on Waymo (Sun et al., 2020). Grid-based map search is 2.7× faster than a general hashmap-based solution thanks to its no-collision property, resulting in a 1.6× end-to-end speedup for mapping. Fusing four small kernels accelerates output construction by 2.1× and brings 1.5× further end-to-end mapping speedup. Finally, simplifying the control logic, loop unrolling and utilizing the symmetry of maps substantially accelerates map search by another 4× and pushes the final end-to-end mapping speedup to **4.6**×.

## 7 Related Work

**Deep Learning on Point Clouds.** Early methods (Chang et al., 2015; Qi et al., 2016; Cicek et al., 2016) first convert point clouds to the dense volumetric representation and apply dense CNNs to extract features. Another line of research (Qi et al., 2017a;b; Li et al., 2018; Wu et al., 2019; Thomas et al., 2019; Wang et al., 2019b) directly performs convolution on the $k$-nearest neighbor or spherical nearest neighbor of each point. Both streams of methods struggle to scale up to large scenes due to large or irregular memory footprint (Liu et al., 2019; 2021). Recent state-of-the-art deep learning methods on point cloud segmentation / detection (Graham et al., 2018; Choy et al., 2019; Tang et al., 2020; Shi et al., 2020; 2021; Yin et al., 2021) are usually based on sparse convolution, which is empirically proven to be able to scale up to large scenes and is the target for acceleration in this paper.

**Point Cloud Inference Engines.** Researchers have extensively developed efficient inference engines for sparse convolution inference. SpConv (Yan et al., 2018) proposes grid-based map search and the gather-matmul-scatter dataflow. SparseConvNet (Graham et al., 2018) 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.

## 8 Conclusion

We present TorchSparse, an open-source inference engine for efficient point cloud neural networks. Guided by two general principles: trade computation for regularity and reduce memory footprint, we optimize matrix multiplication, data movement and mapping operations in sparse convolutions, achieving up to 1.5×, 2.7× and 4.6× speedup on these three components, and up to 1.5-1.6× end-to-end speedup over previous state-of-the-art point cloud inference engines on both segmentation and detection tasks. We hope that our in-depth analysis on the efficiency bottlenecks and optimization recipes for sparse convolution can inspire future research on point cloud inference engine design.

## Acknowledgments

We would like to thank Hanrui Wang and Ligeng Zhu for their feedback on the artifact evaluation. This research was supported by NSF CAREER Award #1943349, Ford and Hyundai. Zhijian Liu and Yujun Lin were partially supported by the Qualcomm Innovation Fellowship.

## REFERENCES


Behley, J., Garbade, M., Milioto, A., Quenzel, J., Behnke, S., Stachniss, C., and Gall, J. SemanticKITTI: A Dataset for Semantic Scene Understanding of LiDAR Sequences. In *IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019.

Caesar, H., Bankiti, V., Lang, A. H., Vora, S., Liong, V. E., Xu, Q., Krishnan, A., Pan, Y., Baldan, G., and Beijbom, O. nuScenes: A Multimodal Dataset for Autonomous Driving. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

Chang, A. X., Funkhouser, T., Guibas, L., Hanrahan, P., Huang, Q., Li, Z., Savarese, S., Savva, M., Song, S., Su, H., Xiao, J., Yi, L., and Yu, F. ShapeNet: An Information-Rich 3D Model Repository. *arXiv*, 2015.

Choy, C., Gwak, J., and Savarese, S. 4D Spatio-Temporal ConvNets: Minkowski Convolutional Neural Networks. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

Cicek, O., Abdulkadir, A., Lienkamp, S. S., Brox, T., and Ronneberger, O. 3D U-Net: Learning Dense Volumetric Segmentation from Sparse Annotation. In *Proc. Medical Image Computing and Computer Assisted Intervention (MICCAI)*, 2016.

Ge, R., Ding, Z., Hu, Y., Shao, W., Huang, L., Li, K., and Liu, Q. 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.

Graham, B., Engelcke, M., and van der Maaten, L. 3D Semantic Segmentation With Submanifold Sparse Convolutional Networks. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

Hu, Y., Ye, Z., Wang, M., Yu, J., Zheng, D., Li, M., Zhang, Z., Zhang, Z., and Wang, Y. FeatGraph: A Flexible and Efficient Backend for Graph Neural Network Systems. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.

Li, Y., Bu, R., Sun, M., Wu, W., Di, X., and Chen, B. PointCNN: Convolution on $\mathcal{X}$-Transformed Points. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

Lin, Y., Zhang, Z., Tang, H., Wang, H., and Han, S. PointAcc: Efficient Point Cloud Accelerator. In *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021.

Liu, Z., Tang, H., Lin, Y., and Han, S. Point-Voxel CNN for Efficient 3D Deep Learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

Liu, Z., Tang, H., Zhao, S., Shao, K., and Han, S. PV-NAS: 3D Neural Architecture Search with Point-Voxel Convolution. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 2021.

Qi, C. R., Su, H., Niessner, M., Dai, A., Yan, M., and Guibas, L. J. Volumetric and Multi-View CNNs for Object Classification on 3D Data. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

Qi, C. R., Su, H., Mo, K., and Guibas, L. J. PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017a.

Qi, C. R., Yi, L., Su, H., and Guibas, L. J. PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017b.

Shi, S., Guo, C., Jiang, L., Wang, Z., Shi, J., Wang, X., and Li, H. PV-RCNN: Point-Voxel Feature Set Abstraction for 3D Object Detection. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

Shi, S., Jiang, L., Deng, J., Wang, Z., Guo, C., Shi, J., Wang, X., and Li, H. PV-RCNN++: Point-Voxel Feature Set Abstraction With Local Vector Representation for 3D Object Detection. *arXiv preprint arXiv:2102.00463*, 2021.

Sun, P., Kretzschmar, H., Dotiwalla, X., Chouard, A., Patnaik, V., Tsui, P., Guo, J., Zhou, Y., Chai, Y., Caine, B., Vasudevan, V., Han, W., Ngiam, J., Zhao, H., Timofeev, A., Ettinger, S., Krivokon, M., Gao, A., Joshi, A., Zhang, Y., Shlens, J., Chen, Z., and Anguelov, D. Scalability in Perception for Autonomous Driving: Waymo Open Dataset. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

Tang, H., Liu, Z., Zhao, S., Lin, Y., Lin, J., Wang, H., and Han, S. Searching Efficient 3D Architectures with Sparse Point-Voxel Convolution. In *European Conference on Computer Vision (ECCV)*, 2020.

Thomas, H., Qi, C. R., Deschaud, J.-E., Marcotegui, B., Goulette, F., and Guibas, L. J. KPConv: Flexible and Deformable Convolution for Point Clouds. In *IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019.

Wang, M., Zheng, D., Ye, Z., Gan, Q., Li, M., Song, X., Zhou, J., Ma, C., Yu, L., Gai, Y., Xiao, T., He, T., Karypis, G., Lin, J., and Zhang, Z. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *arXiv preprint arXiv:1909.01315*, 2019a.

Wang, Y., Sun, Y., Liu, Z., Sarma, S. E., Bronstein, M. M., and Solomon, J. M. Dynamic Graph CNN for Learning on Point Clouds. In *ACM SIGGRAPH*, 2019b.

Wu, W., Qi, Z., and Fuxin, L. PointConv: Deep Convolutional Networks on 3D Point Clouds. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

Yan, Y., Mao, Y., and Li, B. SECOND: Sparsely Embedded Convolutional Detection. *Sensors*, 2018.

Yin, T., Zhou, X., and Krähenbühl, P. Center-based 3D Object Detection and Tracking. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021.

# A Output Coordinates Calculation

Here, we illustrate the output coordinates calculation algorithm for $s > 1$ in sparse convolution. We apply a sliding window on each input point and check whether each candidate output point within the window passes modular and boundary check. If both checks are passed, we add the candidate output point to $\boldsymbol{P}^{\text{out}}$. We finally filter out duplicate coordinates in $\boldsymbol{P}^{\text{out}}$.

**Algorithm 3** Output Coordinates Calculation

```
Input: kernel size K, stride s, input coordinates P^in,
       output coordinates boundary b
Output: output coordinates P^out
  if s = 1 then
    P^out ← P^in
  else
    P^out ← ∅
    for p in P^in do
      # Traverse the neighbors of an input point.
      for δ in Δ^D(K) do
        # Calculate the candidate coordinates.
        u ← p − δ
        # Add output if it passes modular and boundary check.
        if u % s == 0 and u < s · b then
          P^out ← P^out ∪ {u/s}
        end if
      end for
    end for
    # Filter out duplicate coordinates.
    P^out ← Unique(P^out)
  end if
```

# B Adaptive Grouping Algorithm

Here, we provide detailed illustration for the adaptive grouping algorithm. The algorithm is divided into two parts: grouped matrix multiplication (Algorithm 4) and adaptive group search (Algorithm 5).

## B.1 Group Matrix Multiplication

We describe the process of applying the adaptive grouping strategies for each layer in Algorithm 4, which is performed via two steps. First, we maintain two pointers to track the start and the end of the current group. Once $1 - n_{\min}/n_{\max}$ updated by the end pointer exceeds the tolerance of redundant computation $\epsilon$, we return the working group to the groups list and move pointers to start a new group. Second, for each group, we determine if batched matmul is performed on it based on the value of $S$.

## B.2 Adaptive Strategy Search

For each layer, we search for a specific configuration to conduct adaptive grouping (i.e. auto-tune $\epsilon$ and $S$). The tuning

**Algorithm 4** Grouped Matrix Multiplication

```
Input: input features X^in, weights W, maps M,
       redundant computation tolerance ε,
       mm/bmm threshold S
Output: output features X^out
  G ← ∅
  i ← 0
  # Traverse each weight index with unique number of inputs.
  while i < range(⌊Δ^D(K).size/2⌋) do
    n_min ← 0; n_max ← len(M[W_i])
    # Always push the first index to the current group.
    g ← {i}; i ← i + 1
    for j in range(i, ⌊Δ^D(K).size/2⌋) do
      n ← len(M[W_j])
      n_min ← min{n, n_min}, n_max ← max{n, n_max}
      # Push the index to the group if the ratio no larger than ε.
      if 1 − n_min/n_max ≤ ε then
        g ← g ∪ {j}
      else
        # Otherwise return and start a new group.
        break
      end if
    end for
    # Push the returned group to the groups list.
    G ← G ∪ {g}
  end while
  for g in G do
    n_max ← max{len(M[W_i]) for i in g}
    # Pad inputs and apply bmm when workload smaller than S.
    if n_max < S then
      gather input feature matrices F_i with X^in and M[W_i]
      for i ∈ g; pad zeros to each F_i to become length n_max;
      perform batched matrix multiplication between F_{i∈g} and
      W_{i∈g} and then scatter results to corresponding X^out
    else
      # Otherwise apply mm.
      perform Algorithm 2 in main paper with X_i^in for i ∈ g to
      get X^out
    end if
  end for
```

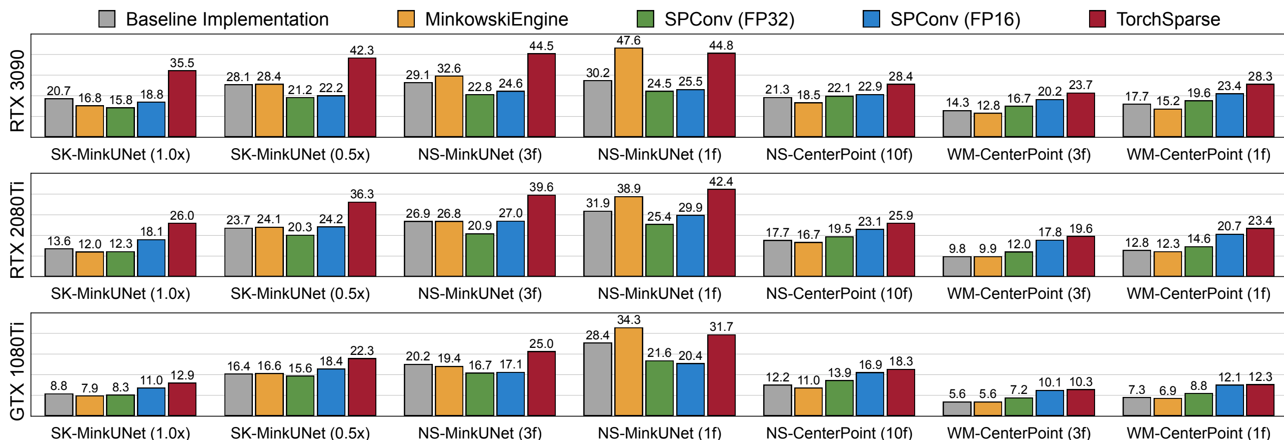



Figure 14: TorchSparse evaluation results in *absolute* values.

algorithm is shown in Algorithm 5, where we enumerate $\epsilon, S$ in a predefined search space (usually $< 1000$ configurations), use Algorithm 4 to perform the matrix multiplication for the target layer, and select the $\epsilon, S$ pair which leads to the smallest average latency.

**Algorithm 5** Adaptive Group Search

**Input:** sampled inputs subset $\boldsymbol{D}$, weights $\boldsymbol{W}$, maps $\mathcal{M}$, redundant computation tolerance search space $\mathcal{S}_a$, `mm`/`bmm` threshold search space $\mathcal{S}_b$

**Output:** selected redundant computation tolerance $\epsilon^*$, `mm`/`bmm` threshold $S^*$

$f \leftarrow$ cost function to compute elapsed time on hardware
$c_{\min} \leftarrow 0$
**for** $\epsilon$ **in** $\mathcal{S}_a$ **do**
  **for** $S$ **in** $\mathcal{S}_b$ **do**
    $c \leftarrow 0$
    **for** $\boldsymbol{X}^{\text{in}}$ **in** $\boldsymbol{D}$ **do**
      $c \leftarrow c + f(\text{run Algorithm 4 with } \boldsymbol{X}^{\text{in}}, \boldsymbol{W}, \mathcal{M}, \epsilon, S)$
    **end for**
    # Update selected config to $\{\epsilon, S\}$ for smaller latency.
    **if** $c_{\min} = 0$ **or** $c < c_{\min}$ **then**
      $c_{\min} \leftarrow c$
      $\epsilon^* \leftarrow \epsilon, S^* \leftarrow S$
    **end if**
  **end for**
**end for**

## C RESULTS DETAIL

We show TorchSparse evaluation results in absolute FPS in Figure 14. TorchSparse is able to run all models in real-time ($> 10$ FPS) on all hardware platforms.