# Exploiting Hardware Utilization and Adaptive Dataflow for Efficient Sparse Convolution in 3D Point Clouds

**Ke Hong**[*1] **Zhongming Yu**[*2] **Guohao Dai**[3] **Xinhao Yang**[1] **Yaoxiu Lian**[3] **Zehao Liu**[1] **Ningyi Xu**[3] **Yuhan Dong**[1] **Yu Wang**[1]

## Abstract

Sparse convolution is the key operator in widely-used 3D point cloud networks. However, due to the high sparsity of voxelized input point cloud data, three main challenges need to be solved for efficient sparse convolution in current 3D point cloud engines: **(1) Memory under-utilization:** the mapping information from input data to weight parameters of 3D point cloud networks is sparse, leading to up to 79.97% redundant memory access and under-utilized memory space; **(2) Computation under-utilization:** previous FGMS (Fused Gather-Matrix-Multiplication-Scatter) operations in sparse convolution are executed sequentially, leading to a GPU computation utilization of only 22.84%; **(3) Input dynamics:** a single and static dataflow in the current point cloud engines cannot always achieve the best performance on different input point cloud data.

To tackle these challenges, we propose PCEngine, an efficient sparse convolution engine for voxel-based 3D point cloud networks. PCEngine proposes a novel coded-CSR (Compress Sparse Row) format to represent the mapping information without redundancy. PCEngine also introduces the indicator-assisted segmented FGMS fusion scheme to fully utilize the computation resources on GPU hardware. PCEngine further deploys a heuristic adaptive dataflow for input dynamics. Extensive experimental results show that PCEngine achieves **1.81**$\times$ and **1.64**$\times$ speedup on average for sparse convolution operation and end-to-end point cloud networks, respectively.

## 1 Introduction

In recent years, 3D point cloud neural network algorithms have achieved significant improvement in scenarios such as autonomous driving, robotics, and AR/VR (Tang et al., 2022; Lin et al., 2021). Even iPhone 14 has been equipped with LiDAR that can generate point cloud data. Moreover, 3D point cloud neural network algorithms have been proven to achieve excellent results on a variety of tasks including object detection and tracking, semantic and instance segmentation (Wang et al., 2019; Wen et al., 2020; Wang et al., 2021). Depending on the format of the point cloud data representation, point cloud neural network algorithms can be roughly classified as voxel-based, point-based, BEV-based, graph-based, etc (Mao et al., 2022). Among these methods, voxel-based networks have achieved state-of-the-art accuracy for a variety of tasks such as semantic segmentation (Choy et al., 2019; Zhu et al., 2021) and object detection (Shi et al., 2020; Deng et al., 2021; Yin et al., 2021).

[*]Equal contribution [1]Tsinghua University [2]University of California, San Diego [3]Shanghai Jiao Tong University. Correspondence to: Guohao Dai <daiguohao@sjtu.edu.cn>, Yu Wang <yu-wang@tsinghua.edu.cn>.

*Proceedings of the 6th MLSys Conference*, Miami, Florida, USA, 2023. Copyright 2023 by the author(s).

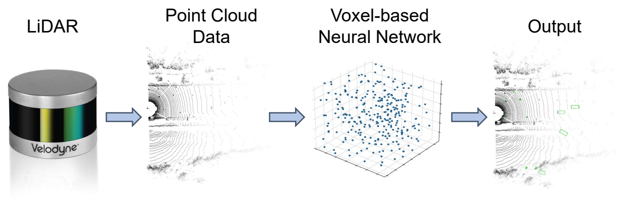


*Figure 1.* An example of generating voxel-based 3D point cloud data and corresponding neural networks.

The typical operation in the voxel-based 3D point cloud networks is sparse convolution. The voxel-based networks divide the 3D features into voxels at corresponding locations based on the coordinates of the input point cloud data, and these voxels are sparsely distributed in a 3D space. Figure 1 shows an example of generating voxel-based 3D point cloud data and corresponding neural networks. Borrowing the idea from the conventional convolution operation in convolutional neural networks, the sparse convolution operation is applied as a core operator for the computation on voxels. Compared with dense image data, these voxels are a set of discrete and highly sparse points which contain position and depth information of objects in the real world. The density of 3D scenes is usually in the range of 0.01% to 1% (Lin et al., 2021). Such high sparsity (low density) leads to problems like poor locality of data access and irregular computation workloads, resulting in poor performance of

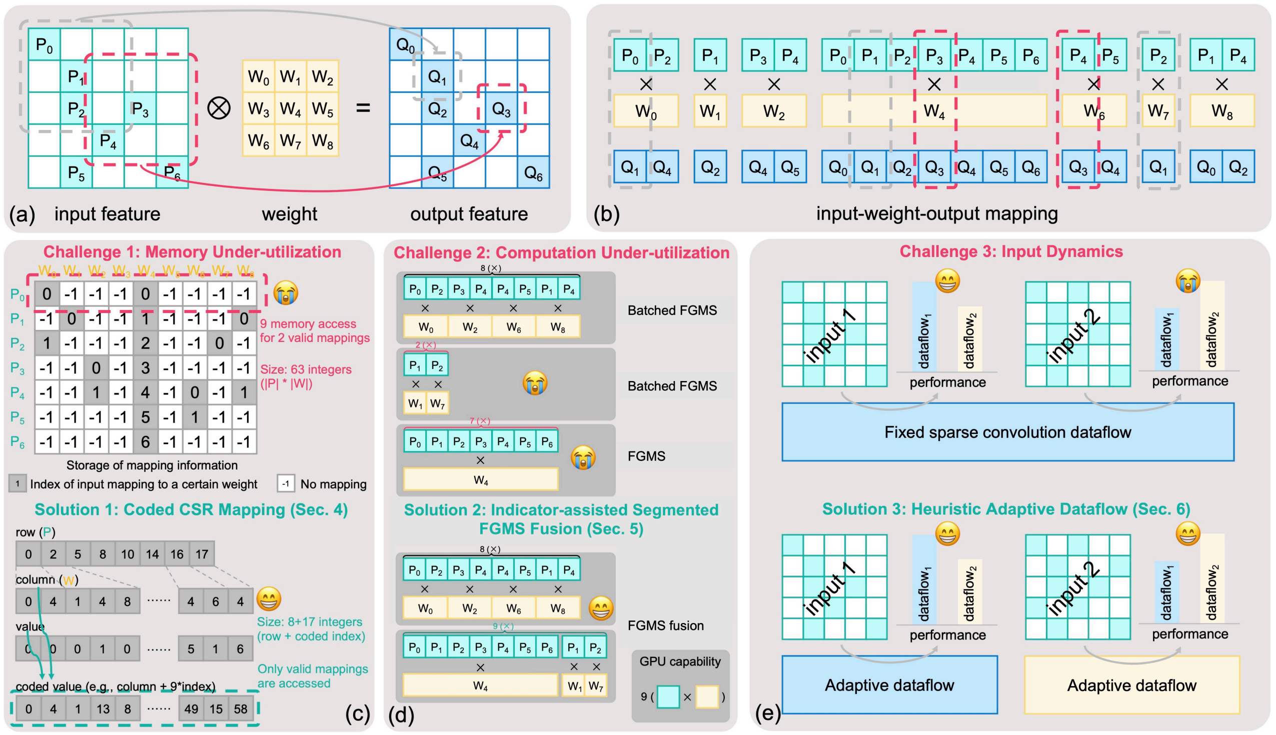

(a) input feature
weight
output feature
(b) input-weight-output mapping
Challenge 1: Memory Under-utilization
9 memory access for 2 valid mappings
Size: 63 integers (|P| * |W|)
Storage of mapping information
Index of input mapping to a certain weight
No mapping
Solution 1: Coded CSR Mapping (Sec. 4)
row (P)
column (W)
value
coded value (e.g., column + 9*index)
Size: 8+17 integers (row + coded index)
Only valid mappings are accessed
(c)
Challenge 2: Computation Under-utilization
Batched FGMS
Batched FGMS
FGMS
Solution 2: Indicator-assisted Segmented FGMS Fusion (Sec. 5)
FGMS fusion
GPU capability
(d)
Challenge 3: Input Dynamics
input 1
input 2
dataflow1
dataflow2
performance
Fixed sparse convolution dataflow
Solution 3: Heuristic Adaptive Dataflow (Sec. 6)
Adaptive dataflow
Adaptive dataflow
(e)

*Figure 2.* Overview of challenges in sparse convolution and solutions in PCEngine. (a) An example of the sparse convolution operation. (b) The mapping among input feature, output feature, and weight. (c) PCEngine stores the mapping using a coded-CSR (Compressed Sparse Row) format to improve memory utilization and reduce redundant memory access. (d) PCEngine introduces the indicator-assisted segmented FGMS (Fused Gather-Matrix-Multiplication-Scatter) fusion technique to fully utilize GPU computation capability. (e) PCEngine proposes the heuristic adaptive dataflow to achieve better performance considering input dynamics.

3D point cloud network computation.

Figure 2(a) shows a typical example of the sparse convolution operation (Graham & van der Maaten, 2017). Sparse output features are obtained from the convolution of sparse input features and dense weights.

Many previous 3D point cloud engines have focused on 3D point cloud network acceleration, such as TorchSparse (Tang et al., 2022), MinkowskiEngine (Choy et al., 2019), SpConv (Yan et al., 2018), and etc (Rusu & Cousins, 2011). SpConv first proposes a Gather-MM (Matrix Multiplication) -Scatter dataflow to organize the sparse convolution computation. MinkowskiEngine utilizes a Fetch-on-Demand dataflow to handle a lighter workload. TorchSparse is one of the state-of-the-art 3D point cloud inference engines, which optimizes irregular computation and data movement in sparse convolution. In order to compute the convolution, these engines first obtain a mapping table based on the relative positions of the inputs, outputs, and weights to find out which input $P$ and weight $W$ the output $Q$ should be computed from, as shown in Figure 2(b). Although many works have proposed several effective techniques for 3D point cloud computation, the sparse characteristic of input voxels makes the sparse convolution operation still suffer from challenges like hardware under-utilization and input dynamics:

- **Memory under-utilization.** Engines like TorchSparse use a matrix (from input features to weights) to store the mapping information (Figure 2(c) top). However, due to the sparsity of input voxels, most elements in the matrix contain invalid mapping information, leading to up to 79.97% redundant memory access and under-utilization of the memory space.
- **Computation under-utilization.** Previous designs organize computation of the sparse convolution operation into multiple FGMS GPU kernels that are executed in a sequence. However, separate FGMS suffers from computation under-utilization (e.g., 22.84% in our experiments) if the total workload is lighter than the GPU capability (Figure 2(d) top).
- **Input dynamics.** Previous designs introduce different dataflows for the sparse convolution operation. These different dataflows imply performance gaps on different input data (Figure 2(e) top). Thus, the fixed dataflow in previous work leads to performance loss considering input dynamics.

To address the above challenges, we propose PCEngine. PCEngine enables efficient processing of voxel-based 3D point cloud networks by exploiting hardware utilization and adaptive dataflow for input dynamics. The contributions of PCEngine include:

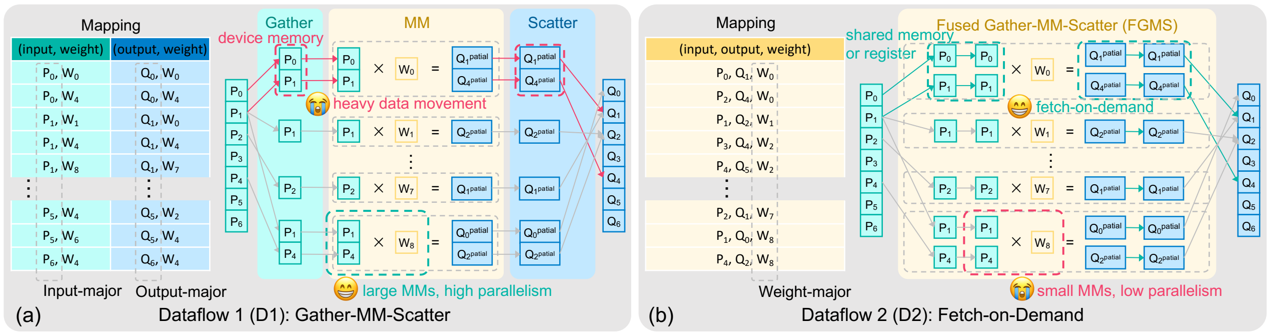

Mapping
(input, weight)
(output, weight)
Input-major
Output-major
Gather
device memory
MM
Scatter
heavy data movement
large MMs, high parallelism
(a)
Dataflow 1 (D1): Gather-MM-Scatter
Mapping
(input, output, weight)
Weight-major
shared memory or register
Fused Gather-MM-Scatter (FGMS)
fetch-on-demand
small MMs, low parallelism
(b)
Dataflow 2 (D2): Fetch-on-Demand

*Figure 3.* Typical dataflow of the sparse convolution operation. (a) The mapping information is stored in two separate tables, and different input features are gathered to a single weight to perform the MM operation, leading to high parallelism while heavy data movement. (b) The mapping information is stored in one table, and the gather, MM, and scatter operations of each weight are fused into a GPU kernel (denoted as FGMS) so that input features are fetched on demand, leading to small data copy overhead while low parallelism of MMs.

- We introduce the coded-CSR (Compressed Sparse Row) format for mapping storage, reducing memory accesses (Figure 2(c) bottom) by 21.18% in our experiments.
- We propose the indicator-assisted segmented FGMS fusion scheme to merge fragmented computations and improve GPU utilization (Figure 2(d) bottom), leading to 1.40× to 1.68× higher GPU utilization.
- We propose a heuristic adaptive dataflow for input dynamics. PCEngine heuristically selects the better dataflow for a certain input, leading to 1.15× to 1.57× average speedup over static dataflow designs.

Extensive experimental results on various benchmarks show that PCEngine achieves **1.81**× and **1.64**× speedup on average for the sparse convolution operation and end-to-end 3D point cloud networks, respectively. The following of this paper is organized as follows. Section 2 introduces backgrounds and preliminaries. An overview of our PCEngine is introduced in Section 3. Three techniques of PCEngine are detailed in Section 4, 5, and 6. PCEngine is evaluated in Section 7. Section 8 introduces related works, and Section 10 concludes the paper.

## 2 Backgrounds and Preliminaries

### 2.1 Definition of Sparse Convolution

A typical example of the sparse convolution operation is shown in Figure 2(a). Let $P_i$ represent the feature vector of the input data $\mathrm{P}_i$, $Q_j$ represent the feature vector of the output data $\mathrm{Q}_j$, and $W_k$ represent the weight matrix of weight $\mathrm{W}_k$. In this example, we have $k \in \{0..8\}$ for this 3×3 weight. Then we have:

$$Q_j = \sum_{k\in\{0..8\}} W_k P_{j(k)}, P_j \neq 0. \tag{1}$$

Here, $P_{j(k)}$ represents the input feature vector of the input data $\mathrm{P}_{j(k)}$, where the offset from $P_{j(k)}$ to $P_j$ is the same as the offset from $W_k$ to $W_4$ (the center of the weight matrix). The sparsity of input and output data in the example of Figure 2(a) and Equation (1) remains the same, because we only calculate $Q_j$ when $P_j \neq 0$. Such type of the sparse convolution is called the submanifold sparse convolution (Graham et al., 2018). Another typical type of the sparse convolution is more general, where the constraint $P_j \neq 0$ is removed. Such type of the sparse convolution makes the non-zero elements grow rapidly.

### 2.2 Typical Dataflow of Sparse Convolution

Based on the definition of the sparse convolution operation, the mapping information among input features, weights, and output features can be generated, shown in Figure 2(b). Such mapping information indicates the dependency between an output feature vector $Q_j$ and an input feature vector $P_i$ or weight parameter $W_k$. There are two ways to store the mapping information. The first way is storing the mapping from input features to weight parameters and output features to weight parameters in two separate tables, while another way is using one table to store the tuples of (input, output, weight). These two ways are utilized in different dataflows of the sparse convolution operation.

#### 2.2.1 *Dataflow 1 (D1): Gather-MM-Scatter*

The Gather-MM-Scatter dataflow (D1) is shown in Figure 3(a). D1 constructs two mapping tables to store the mapping information from input features to weights, and output features to weights, respectively. As is introduced in TorchSparse (Tang et al., 2022), the separate mapping enables data reuse in both gather and scatter operations. Based on the $(\mathrm{P}_{j(k)}, \mathrm{W}_k)$ mapping, for each $\mathrm{W}_k$ all input feature vectors $P_{j(k)}$ are gathered together to form a feature

*Table 1.* Notations in this paper

| Notation | Meaning |
|---|---|
| $P_i$ | feature vector of input data $\mathrm{P}_i$ |
| $Q_j$ | feature vector of output data $\mathrm{Q}_j$ |
| $W_k$ | the $k$-th weight in the convolution kernel |
| $N^{\mathrm{in}}$ | the number of input features |
| $N^{\mathrm{out}}$ | the number of output features |
| $K$ | the number of weights in the convolution kernel |

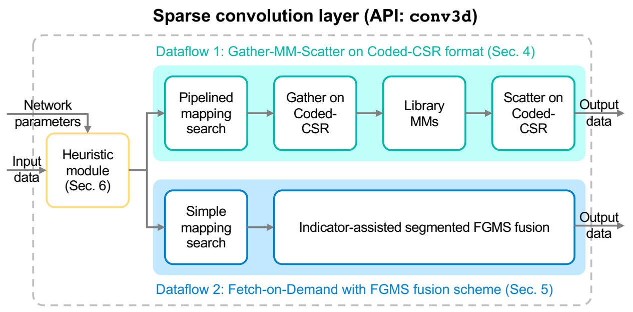

Sparse convolution layer (API: conv3d)
Dataflow 1: Gather-MM-Scatter on Coded-CSR format (Sec. 4)
Network parameters
Input data
Heuristic module (Sec. 6)
Pipelined mapping search
Gather on Coded-CSR
Library MMs
Scatter on Coded-CSR
Output data
Simple mapping search
Indicator-assisted segmented FGMS fusion
Output data
Dataflow 2: Fetch-on-Demand with FGMS fusion scheme (Sec. 5)

*Figure 4.* The system overview of PCEngine. Two dataflows are wrapped and dynamically selected. The coded-CSR format and the indicator-assisted segmented FGMS fusion scheme are introduced to Gather-MM-Scatter dataflow and Fetch-on-Demand dataflow, respectively.

matrix. The gathered feature matrices of all $\mathrm{W}_k$ are stored in a buffer in the GPU device memory. Then during the MM stage, the feature matrix corresponding to each $\mathrm{W}_k$ is multiplied with the weight matrix $W_k$ sequentially, deriving the partial sum vectors $Q_j^{\text{partial}}$. Further, according to the ($\mathrm{Q}_j$, $\mathrm{W}_k$) mapping, these partial sum vectors are scattered and accumulated to the corresponding output feature vectors $Q_j$. The MMs can be executed in high parallelism using libraries like cuBLAS (Nvidia's BLAS library), making D1 efficient in sparse convolution computation of scale.

### 2.2.2 *Dataflow 2 (D2): Fetch-on-Demand*

Another dataflow is the Fetch-on-Demand scheme (D2). D2 utilizes a combined mapping ($\mathrm{P}_{j(k)}$, $\mathrm{Q}_j$, $\mathrm{W}_k$). Instead of executing MMs after gathering all the required input feature vectors into a buffer, D2 fuses the feature fetching process into the MM computation. Specifically, for each $\mathrm{W}_k$ the corresponding feature vectors, $P_{j(k)}$ are fetched into the shared memory or registers directly and computed to derive the partial sum vectors $Q_j^{\text{partial}}$. Once the MM computation is finished, the partial sum vectors are directly added to the output features $Q_j$. The computation corresponding to each $\mathrm{W}_k$ is defined as a Fused Gather-MM-Scatter (FGMS for short) operation in the following. In D2, no buffer is needed to store the intermediate results, and hence the data movement cost is significantly reduced. However, the lack of a pre-fetch stage prevents MMs from being organized into large-scale computations. The low parallelism makes the MM stage become the bottleneck in D2, especially when the size of the input feature grows large. As a result, the reduced data movement of D2 leads to better performance with a light workload, and the low parallelism of D2 leads to performance degradation as the workload grows heavier.

## 3 Overview of PCEngine

The system overview of PCEngine is illustrated in Figure 4. The sparse convolution operation is exposed as a `conv3d` API in upper-layer frameworks. Two typical types of dataflow introduced in Section 2.2 and Figure 3 are both wrapped into the API. Specifically, PCEngine introduces the coded-CSR format for mapping to the Gather-MM-Scatter dataflow, which will be detailed in Section 4. A pipelined mapping search is utilized to build the coded-CSR format mapping. Then, the following gather and scatter operations are designed corresponding to the coded-CSR mapping format. For the Fetch-on-Demand dataflow, a simple mapping search operation is applied. To fully utilize the parallelism of GPU hardware, PCEngine proposes the indicator-assisted segmented FGMS fusion instead of separate FGMSs (Choy et al., 2019) to derive the output, which is introduced in Section 5. In order to enable adaptability to input dynamics, PCEngine takes features of input data and point cloud networks to a heuristic module to select different dataflow heuristically and dynamically, introduced in Section 6.

## 4 Coded-CSR Mapping

**Motivation.** In Gather-MM-Scatter dataflow, the gather and scatter operations take up to 40% of the total execution time (Tang et al., 2022), becoming significant parts in accelerating the sparse convolution operation. Previous designs like TorchSparse propose to use an input-major mapping for the gather operation and an output-major mapping for the scatter operation (shown in green and blue in Figure 3(a), respectively). Thus, an input feature is only read once to different weights, and an output feature is only written once to the device memory after accumulation. In that way, both input-level and output-level data reuse are exploited.

However, storing and representing this input-major and output-major mapping information fails to make full use of memory in previous designs like TorchSparse. Particularly, the input mapping is recorded as a $(P_{j(k)}, W_k)$ matrix in TorchSparse. However, due to the sparsity of the point cloud data, a specific input is not mapped to all possible weights. The profiling result indicates that for the KITTI dataset (Geiger et al., 2012), up to 79.97% elements of the mapping matrix are empty with a $3\times3\times3$ kernel, leading to redundant memory access in memory-intense gather and scatter operations.

**Analysis.** Here we take the input mapping as an example (the analysis methodology for the output mapping is similar), two types of data are accessed, i.e., the input features and the mapping information. The input-major mapping scheme in TorchSparse eliminates the redundancy of loading input features. However, due to the sparsity of mapping from input features to weights, redundancy still remains in loading the mapping information. Let $IF$ denote the data access amount of the input features, $MP$ denote the data loading amount of the mapping information, and $density$ denote the ratio of valid elements (not -1) in the mapping matrix from the input features to weights. Thus, the ratio of

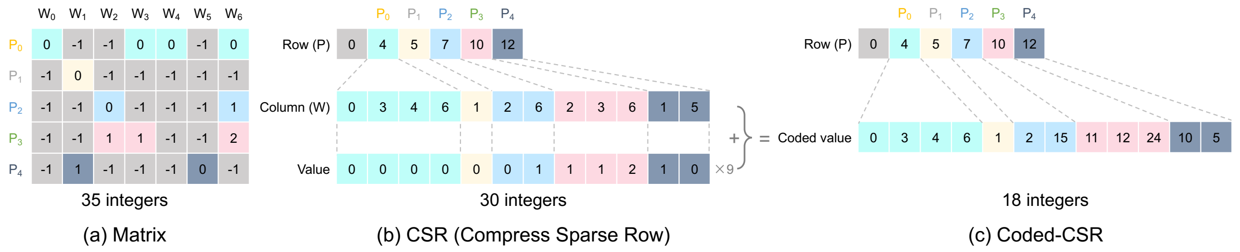


*Figure 5.* An example of the code-CSR format. (a) The sparse mapping matrix from input features to weights. (b) The CSR format of the matrix. (c) The coded-CSR format encodes the column array and the value array in the CSR format with a scaling factor of 9.

redundant data loading, $R_{redundancy}$, can be formulated as:

$$\begin{aligned} R_{redundancy} &= \frac{(1-density)\times MP}{IF+MP} \\ &= \frac{1-density}{IF/MP+1}. \end{aligned} \tag{2}$$

We can find the ratio of redundancy increases when: (1) $density$ decreases (e.g., less valid elements are in the mapping matrix); (2) $IF$ decreases (e.g., the channel size of input features decreases). We profile typical point cloud samples, and $R_{redundancy}$ can be up to 38.26% in a $(c^{\text{in}}, c^{\text{out}}) = (4, 16)$ sparse convolution layer, becoming negligible if only the redundancy of loading input features is removed in previous designs.

**Challenge.** We need to apply a proper sparse matrix format to the mapping storage so that the redundant memory access can be removed. Moreover, the memory usage of the input-major and output-major mapping is much higher than the kernel-major mapping. Thus, the storage format should be light in memory usage. Potentially, the format conversion brings extra latency to the system, and a specific design needs to be proposed to reduce the overhead.

**Insight.** Intuitively, the Compress Sparse Row format, CSR, for sparse matrices can be applied to store the sparse mapping instead of using the matrix in a dense representation way. Valid elements in the mapping matrix are stored using three arrays, a row array, a column array, and a value array. Specifically, the input features are rows and weights are columns, and the value array stores the value of valid elements in the mapping matrix. The column array stores indices of weights, which are no larger than the size of the sparse convolution kernel size $K$ (e.g., 9 for a 3×3 kernel, and 27 for a 3×3×3 kernel). Thus, our key insight is, **the value in the column array is relatively small, and can be encoded into the the value array.**

**Approach.** We leverage the advantage of the CSR format to store the mapping matrix in a compressed way. Based on the insight that the column array can be encoded into the value array, we further generate a coded value array from the column array and the value array in the original CSR format. Let $a$ and $b$ represent a value in the column array and its corresponding value stored in the value array, we have $a < K$ where $K$ is the total number of weights in the weight kernel. Thus, we take a simple encoder function:

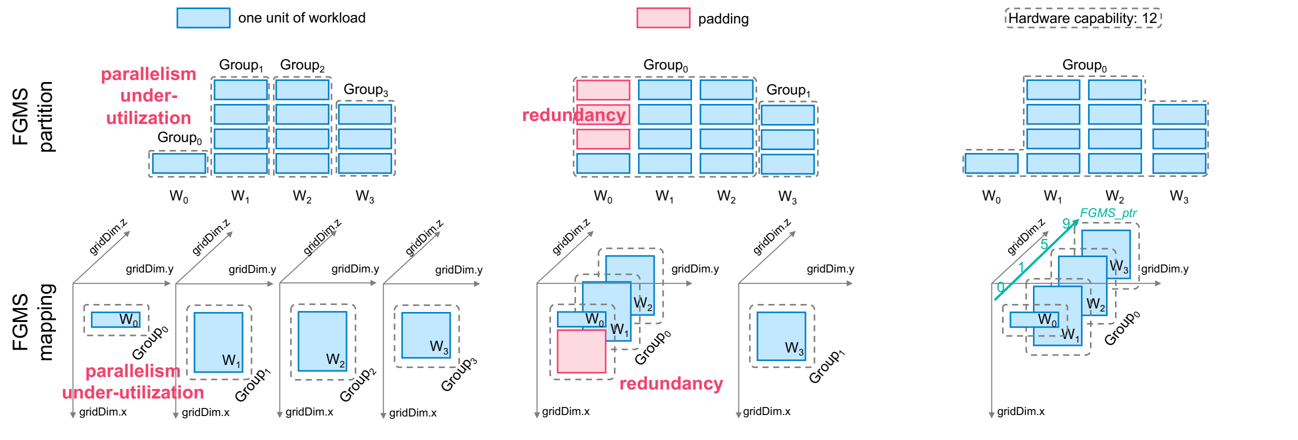
$$c = a + K \times b. \tag{3}$$

Here, $c$ represents the corresponding value in the coded value array. Because we have $a < K$, we can easily get $a$ and $b$ from $c$ with a simple modulus operation. The overhead of the coded-CSR format mapping includes the format conversion and the encoding-decoding process. The format conversion and the encoder are implemented into the mapping search operation, and the decoder is fused into both gather and scatter operators. To alleviate the overhead in the mapping search, we decouple the dataflow and design a pipeline for the operators, which is detailed in Appendix C. Since gather and scatter operators are both memory-intense GPU kernels, the computation resources are relatively abundant. As a result, the fusion of the decoder into gather and scatter operators hardly causes any extra latency.

**Example.** Figure 5 shows a simple example of implementing the coded-CSR format. Figure 5(a) stores the mapping information from 5 input features to 7 weights. The non-negative value in the matrix represents the index of input features multiplied by a certain weight, while the "-1" value represents an invalid mapping. Figure 5(b) represents a typical CSR representation for the matrix in Figure 5(a). Because there are 7 weights in the matrix, the maximum value in the column array is smaller than 7. For general purposes, the scaling factor $K$ in Equation (3) can be an arbitrary integer no smaller than $K$. Here, we set the scaling factor to be 9 (>7), and encode the column and the value array into the coded value array in Figure 5(c). Compared with 35 integers stored in the matrix, only 18 integers are stored in the row array and the coded value array. The coded-CSR format significantly improves memory space utilization and eliminates redundant access to invalid mappings.

## 5 Indicator-Assisted Segmented FGMS Fusion

**Motivation.** FGMS operations are dominant in the fetch-on-demand dataflow, taking up to 75% runtime. Due to the sparsity and the irregularity of the point cloud, different input features $P_i (i \in \{0..N^{\text{in}} - 1\})$ are gathered and



(a) Separate FGMS with parallelism under-utilization (b) Batched FGMS with redundant padding (c) Indicator-assisted segmented FGMS fusion

*Figure 6.* Examples of different FGMS schemes. One unit of workload represents the computation on a valid mapping ($\mathrm{P}_{j(k)}$, $\mathrm{Q}_j$, $\mathrm{W}_k$), i.e., to load $P_{j(k)}$, multiply $P_{j(k)}$ by $W_k$, and add $Q_j^{\text{partial}}$ to $Q_j$. (a) Using small groups for the separate FGMS, leading to computation parallelism under-utilization. (b) Using paddings for the batched FGMS, leading to computation redundancy. (c) The indicator-assisted segmented FGMS fusion executes FGMSs of different sizes, eliminating computation redundancy and improving hardware utilization.

weighted with different $W_k (k \in \{0..K-1\})$. Naturally, we can not piece together the FGMS operations of different $W_k$ to derive a larger FGMS. As a result, those separate FGMS operations are too light for hardware utilization. Although those separate FGMSs can not be fused from the mathematical perspective, they can be executed simultaneously on GPU with specific mapping designs. In this way, the computation utilization of GPU hardware can be improved.

**Analysis.** As mentioned in Section 2.2, it is significant to improve the hardware parallelism during the FGMS stage in D2. According to the experiments, the workload of a separate FGMS is far from GPU capacity. On ModelNet40, the average GPU utilization of FGMSs is only 22.84% with D2 on an Nvidia RTX 3090 GPU. In sparse convolution practice, TorchSparse proposes to batch MMs together for computation in D1. The idea of batching segmented computations together can be applied to D2 as well. However, the batching technique is designed to batch computations of the same size, which is not the case with sparse convolution for point clouds. Thus, the application of batched FGMS in sparse convolution needs extra paddings and causes computation redundancy. Some adaptive strategies for batched MMs (Tang et al., 2022) can be used to reduce the redundancy, by only batching computations of similar sizes together. However, the redundancy avoidance principle causes small FGMSs not to be batched with neighboring FGMSs due to size difference, which still leads to the issue of hardware under-utilization. The profiling results on ModelNet40 (Wu et al., 2015) with an Nvidia RTX 3090 GPU shows that the average GPU utilization of batched FGMSs is only 30.48%.

**Challenge.** As our analysis has shown, previous schemes such as separate FGMS and batched FGMS suffer from various drawbacks. The computation of a separate FGMS is far from GPU capacity, and the batched FGMS fails to improve the utilization without bringing redundancy. Thus, a fusion scheme that fully considers the specific workload of sparse convolution is needed for FGMS operations to be efficiently executed on GPU.

**Insight.** Drawing inspiration from works like CUTLASS (Nvidia, 2023) and ByteTransformer (Zhai et al., 2023), we discover that a group of FGMS operators can be fused together without any padding, thereby eliminating the redundancy issue and increasing GPU utilization. The batching technique pads different workloads into equal sizes so that each workload is easily recognized on the hardware. Thus, the key insight is that **no padding is needed if the workload of different FGMS can be distinguished during computation through some other method.** To achieve that goal, we propose a method called indicator-assisted fusion, which is detailed as follows.

**Approach.** In order to index the workload for each FGMS without padding, we propose the indicator-assisted segmented FGMS fusion scheme. Based on the fact that only one FGMS dimension (i.e., the input size dimension) varies in sparse convolution, we propose to achieve the FGMS scheduling based on an indicator. The fused FGMS utilizes an auxiliary array, $FGMS\_ptr$, to indicate the mapping table address for each FGMS in the memory, shown in Figure 6(c). Each FGMS is enabled to index and gather the corresponding input features (or scatter the partial sums) based on its own part of the mapping table independently. In that way, the computation redundancy brought by the size difference is removed, and the separate FGMSs can be continuously mapped to the computation units until sufficient hardware utilization is reached.

**Example.** Figure 6 shows the example of the indicator-assisted segmented FGMS fusion scheme and compares it with the separate FGMS and the batched FGMS schemes.

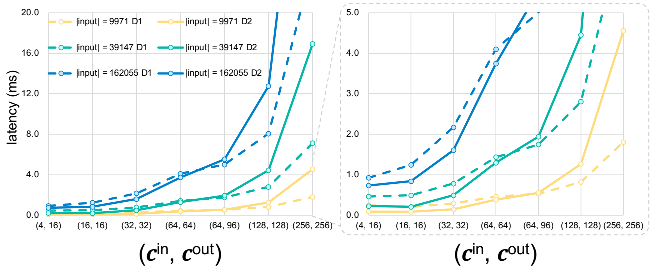


*Figure 7.* Performance comparison between D1 and D2 with different numbers of input voxels and sizes of input/output channels.

The example contains 4 weights ($W_0$ to $W_3$) with different workloads (e.g., 1 unit for $W_0$ and 4 for $W_1$). Figure 6(a) divides weights into 4 groups while the parallelism of GPU capacity is not fully utilized. Figure 6(b) divides weights into 2 groups, and pads the workload for $W_0$, leading to computation redundancy. Figure 6(c) depicts our indicator-assisted segmented FGMS fusion scheme. For weights of different workloads, the $FGMS\_ptr$ is introduced to distinguish the FGMS workload of each weight. The $\text{group}_0$ in Figure 6(c) can fully utilize GPU parallelism capacity by using our proposed scheme.

## 6 Heuristic Adaptive Dataflow

**Motivation.** Input dynamics lead to differentiated relative performance among different sparse convolution dataflows. For two typical types of dataflow introduced in Section 2.2, D1 (Gather-MM-Scatter) makes better utilization of GPU hardware, which leads to better performance in sparse convolution computation of scale. D2 (Fetch-on-Demand) eliminates frequent input/output feature copies in GPU device memory, which is better for small-scale problems. Thus, adaptive dataflow is required to achieve better performance considering input dynamics.

**Analysis.** The input dynamics have great impacts on the relative performance between D1 and D2. The major input characteristics include the input size and the input-output channel size pair $(\boldsymbol{c}^{\text{in}}, \boldsymbol{c}^{\text{out}})$. To visualize the influence of each input characteristic, we test the latency of a sparse convolutional layer with different input characteristics and dataflows on an Nvidia RTX 2080 GPU. As shown in Figure 7, D2 enables the sparse convolution to perform faster than D1 when the channel sizes and the input size are both small. However, the performance of D2 degrades rapidly as the channel sizes expand, and a larger input size accelerates the degradation. The profiling results indicate that we need to be cautious in selecting D2 for computation, otherwise, the latency brought by improperly using D2 can significantly hurt the system performance.

**Challenge.** We observe that the naive heuristic principle brings little benefit to the system. Specifically, the inappropriate design causes redundant mapping searches in the

**Algorithm 1** Dataflow Heuristic Algorithm

```
Input: layer set $\mathcal{L}$, input channel list $\boldsymbol{c}^{\text{in}}$, output channel list $\boldsymbol{c}^{\text{out}}$, stride list $\boldsymbol{s}$, channel threshold $B$
Output: D2 layer candidate set $\mathcal{M}$
$\mathcal{M} \leftarrow \mathcal{L}$; $\boldsymbol{s}^t \leftarrow \mathbf{1}$;
# Build up the temporary stride list $\boldsymbol{s}^t$
for $i, s_i$ in enumerate($\boldsymbol{s}$) do
  if layer $i$ is not the first layer then
    $s_i^t \leftarrow s_i \times s_{i-1}^t$
  end if
end for
# Traverse all the layers
for $i$ in range($|\mathcal{L}|$) do
  if $i$ not in $\mathcal{M}$ then
    continue
  end if
  if $\max(c_i^{\text{in}}, c_i^{\text{out}}) > B$ then
    $\mathcal{M} \leftarrow \mathcal{M} \setminus i$
    continue
  end if
  # Check the layers within the same group
  for $j \in \{k \in \mathcal{L} | s_k = s_i\} \cap \{k \in \mathcal{L} | s_k^t = s_i^t\}$ do
    if $\max(c_j^{\text{in}}, c_j^{\text{out}}) > B$ then
      $\mathcal{M} \leftarrow \mathcal{M} \setminus \{i, j\}$
      break
    end if
  end for
end for
```

network, which is quite time-consuming. Based on the fact that different mapping patterns are used in D1 and D2 (as shown in Section 2.2), an extra mapping search is demanded whenever neighboring layers employ different dataflows. Thus, the algorithm is supposed to be carefully designed to adapt to the user-defined network.

**Insight.** Based on our analysis, the channel size pair $(\boldsymbol{c}^{\text{in}}, \boldsymbol{c}^{\text{out}})$ is the key factor to influence the relative performance between different dataflows. Our key insight is, **D1 achieves better performance for large-scale problems, while D2 is better for small-scale problems.** A channel size threshold $B$ can be introduced to determine how large the channel size is when we stop using D2. The other insight is that **the sparse convolution layers with the same mapping can be found based on the network structure.** On the basis of that, we can separate the sparse convolution layers of a network into groups. The layers within the same group share the same mapping, and the dataflow selection decision can be made on a group basis to avoid additional mapping searches caused by heuristics.

**Approach.** The detailed algorithm is presented in Algorithm 1. First, we build up the temporary stride list by enumerating the stride $s$ of each layer in order. The temporary stride $s^t$ of a layer is the product of all the former layer strides, and the layers of the same $(s, s^t)$ pair use the same mapping. Based on that, we divide the layers into groups. A candidate list $\boldsymbol{m}$ including the layers that potentially apply D2 is maintained, and the list is initialized with all layers. In each group, $\max(c_i^{\text{in}}, c_i^{\text{out}})$ of every layer $i$ is compared with the channel threshold $B$. If the max channel size of any layer in the group exceeds $B$, we delete the group from the candidate list. Finally, the layers in the groups left as candidates are arranged to apply D2 to the convolution.

**Example.** Figure 8 shows an example of applying the heuristic adaptive dataflow. PCEngine can apply different dataflows for different sparse convolution layers in a point cloud network. Specifically, we set $B = 64$ and apply the proposed heuristic algorithm to SparseResNet. SparseResNet contains 21 sparse convolution layers. Those layers are divided into eight groups according to the mapping usage, and the heuristic algorithm arranges the layers in the first four groups to use D2.

## 7 Evaluation

### 7.1 Experiments Setup

#### *7.1.1 Baselines*

PCEngine is implemented with a C++ and CUDA backend based on CUDA 11.1, and a PyTorch-based front-end based on PyTorch 1.10.0. We compare the inference performance with two state-of-the-art designs:

- TorchSparse v2.0.0 (Tang et al., 2022) is one of the mainstream sparse convolution engines on GPUs, which optimizes the Gather-MM-Scatter dataflow and outperforms the former works like MinkowskiEngine.
- SpConv v2.2.3 (Traveller59, 2022) is the updated version of the work in (Yan et al., 2018), which computes the sparse convolution in the implicit GEMM (General Matrix Multiplication) way similar to the dense convolution, and currently achieves the best performance in some circumstances.

We use the same downsampling principle defined by TorchSparse (Tang et al., 2022) in the strided sparse convolutional layers for all engines. All the experiments are conducted under FP32 precision.

#### *7.1.2 Benchmarks*

We test the performance of PCEngine and other engines on two typical 3D point cloud networks:

- SparseResNet (Choy et al., 2019) is a common sparse convolution network with the ResNet (He et al., 2016) structure, which contains 21 sparse convolutional layers.
- MinkUNet (Choy et al., 2019) adopts the U-Net (Ronneberger et al., 2015) structure, which contains 42 sparse convolution layers.

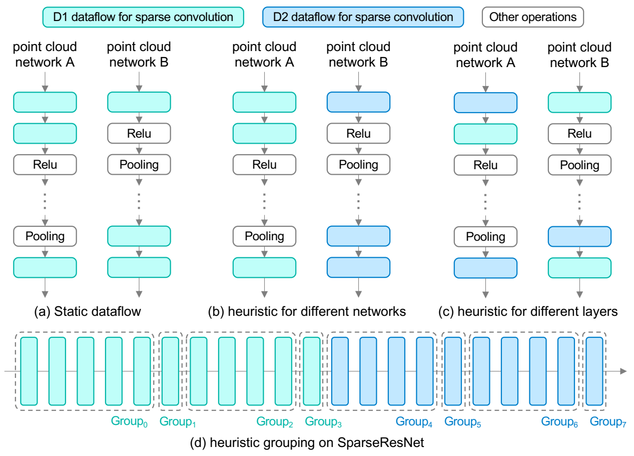

D1 dataflow for sparse convolution
D2 dataflow for sparse convolution
Other operations
point cloud network A
point cloud network B
Relu
Pooling
(a) Static dataflow
(b) heuristic for different networks
(c) heuristic for different layers
Group$_0$
Group$_1$
Group$_2$
Group$_3$
Group$_4$
Group$_5$
Group$_6$
Group$_7$
(d) heuristic grouping on SparseResNet

*Figure 8.* Examples of heuristic adaptive dataflow. (a) The static dataflow. (b) Heuristics for different networks. (c) Heuristics for different layers. (d) Heuristic grouping on SparseResNet.

#### *7.1.3 Datasets*

We use 3 datasets for different point cloud applications:

- ModelNet40 (Wu et al., 2015) is a comprehensive collection of 3D CAD models for objects from 40 categories. Input voxels from this dataset are of 1%-level density.
- S3DIS (Armeni et al., 2016) is an indoor scene 3D point cloud dataset, which is generally used for 3D semantic segmentation. The points from a whole scene are sampled and voxelized into input data of 1%-level density.
- KITTI (Geiger et al., 2012) is a common algorithm evaluation dataset in autonomous driving, including real data collected in outdoor scenarios. Input data are of 0.01%-level density after voxelization, and this dataset is generally used for 3D object detection and tracking.

#### *7.1.4 Platforms*

The evaluation experiments are conducted both on an Nvidia RTX 2080 GPU and an Nvidia RTX 3090 GPU. Note that Nvidia RTX 3090 GPU enables the usage of TF32 precision in MMs to accelerate computation for all engines.

### 7.2 Overall Performance

**End-to-end Performance.** We compare the end-to-end performance of PCEngine and two state-of-the-art engines, as illustrated in Figure 9(a). The results indicate that, on average, PCEngine achieves **1.64**$\times$ and 1.23$\times$ speedup over TorchSparse and SpConv, respectively. Notably, PCEngine achieves a higher 1.73$\times$ speedup over TorchSparse on RTX 3090. This can be attributed to PCEngine's superior GPU utilization, which results in higher performance gains on more powerful GPUs.

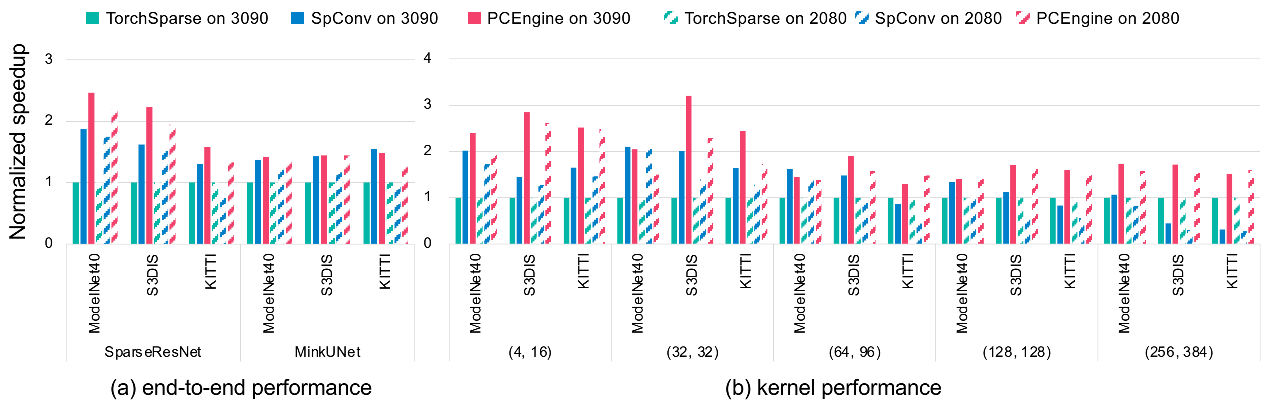


*Figure 9.* Performance comparison between PCEngine and state-of-the-art point cloud engines. (a) End-to-end performance. (b) Sparse convolution kernel performance.

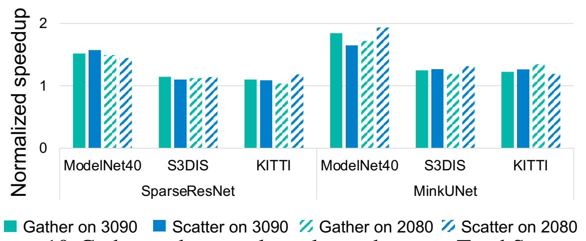


*Figure 10.* Gather and scatter kernel speedup over TorchSparse.

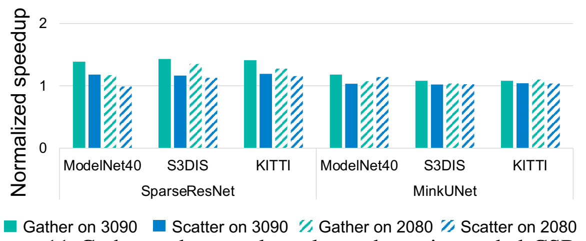


*Figure 11.* Gather and scatter kernel speedup using coded-CSR.

**Kernel Performance.** We further compare the kernel performance of the sparse convolution operation. We vary the sizes of input and output channels to prove the effectiveness of PCEngine. We set five different channel sizes ($c^{\text{in}}, c^{\text{out}}$), and the results are shown in Figure 9(b). PCEngine achieves **1.81**$\times$ and 1.76$\times$ speedup on average for the sparse convolution operation over TorchSparse and SpConv, respectively. In particular, PCEngine achieves up to 3.20$\times$ speedup over TorchSparse on small channel size (i.e., (4,16) on RTX 3090) because PCEngine removes redundant access of mapping information, which is negligible for features of small sizes. PCEngine also achieves up to 5.56$\times$ speedup over SpConv on large channel size (i.e., (256, 384) on RTX 3090) because SpConv is not optimized for large feature sizes.

**Gather and Scatter Performance.** The gather and scatter operations take up to 40% of the total execution time for the Gather-MM-Scatter dataflow, and PCEngine proposes coded-CSR mapping to reduce redundant mapping loadings for these two kernels. We compare the performance of these two kernels with TorchSparse in Figure 10, as only TorchSparse adopts the Gather-MM-Scatter dataflow. PCEngine outperforms TorchSparse in all cases, achieving 1.31$\times$ and 1.33$\times$ average speedup for gather and scatter kernels, respectively.

## 7.3 Benefits of PCEngine Designs

**Coded-CSR.** Figure 11 shows the speedup brought by the coded-CSR format. The coded-CSR format eliminates redundant mapping information access, leading to 1.21$\times$ and 1.10$\times$ speedup for the gather and scatter kernels. Moreover, we profile the average data transfer for these two kernels. The coded-CSR format leads to 21.18% less memory access on average for two kernels.

**Indicator-assisted Segmented FGMS Fusion.** Figure 12 shows the performance comparison of separate FGMSs, batched FGMSs and indicator-assisted segmented FGMS fusion by PCEngine. The proposed scheme in PCEngine outperforms the other two schemes in all cases, with an average speedup of 1.75$\times$ and 1.41$\times$ over two schemes. More importantly, the proposed scheme improves the GPU utilization by 1.68$\times$ and 1.40$\times$, respectively.

**Heuristic Adaptive Dataflow.** Figure 13 demonstrates the benefits of our heuristic adaptive dataflow. The end-to-end performance is normalized to the heuristic adaptive dataflow. The results reveal that a static dataflow (Gather-MM-Scatter or Fetch-on-Demand) is slower than the heuristic adaptive dataflow. Our heuristic design achieves 1.15$\times$ and 1.57$\times$ speedup over two static designs on average, indicating that applying heuristic adaptive dataflow does accelerate 3D point cloud network processing.

## 7.4 Threshold Choice

PCEngine heuristically utilizes different dataflows for different channel sizes by setting the threshold $B$. Figure 14 shows the influence of choosing different thresholds on the end-to-end latency, running SparseResNet using RTX 3090. As we can see, $B > 64$ is better for the ModelNet40 and KITTI datasets, while $B = 32$ is better for the S3DIS

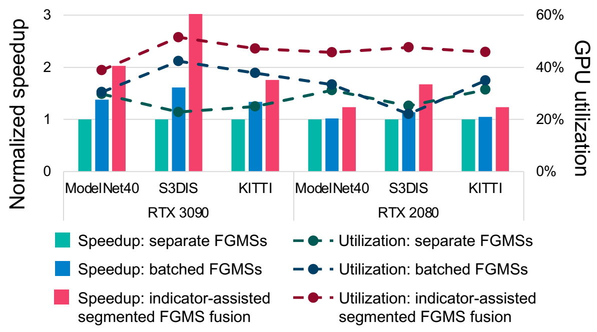


*Figure 12.* Benefits of the indicator-assisted segmented FGMS fusion scheme.

dataset. We set $B = 64$ for evaluation without loss of generality. The results indicate that the performance of PCEngine can be further improved. We do not intentionally tune the performance of PCEngine in previous results.

## 8 Related Works

**Point Cloud Networks.** (Wu et al., 2015; Maturana & Scherer, 2015) leverage a regular 3D voxel grid to represent the point cloud and perform 3D convolution on the voxels. But the voxel grid resolution is strictly limited in these methods as the memory and computational cost grow cubically with the resolution expanding. (Riegler et al., 2017; Wang et al., 2017) perform convolution based on the octree structure of the 3D shape, which reduces the memory and computational cost to support high resolution. (Liu et al., 2019; Boulch, 2019; Wu et al., 2019) apply the convolution directly on the 3D points by performing a weighted sum over a local subset according to the center point.

**Point Cloud Engines.** The convolution operation for sparse input data is proposed in (Graham, 2015). In order to maintain the sparsity of features in sparse convolution networks, (Graham & van der Maaten, 2017) proposes submanifold sparse convolutions to constrain the non-zero elements through the network to a certain spatial range. SparseConvNet (Graham, 2015) and SpConv (Yan et al., 2018) are frameworks to support sparse convolution computation. By constructing mapping, effective computations are extracted, and sparse convolutions are converted into dense matrix multiplication operations. MinkowskiEngine (Choy et al., 2019) achieves parallel acceleration for the above Gather-MM-Scatter dataflow by using CUDA libraries. In addition, it also proposes Fetch-on-Demand dataflow to reduce data movement and support small-scale computing. TorchSparse (Tang et al., 2022) optimizes irregular computation and data movement in sparse convolution calculations, which greatly improves the performance of Gather-MM-Scatter dataflow. Recently, another dataflow, implicit GEMM (Traveller59, 2022), has been proposed, which computes sparse convolution in a dense way to improve computational parallelism.

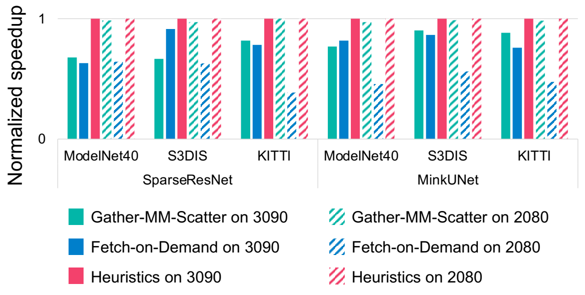


*Figure 13.* Performance comparison between the heuristic dataflow and different static dataflow.

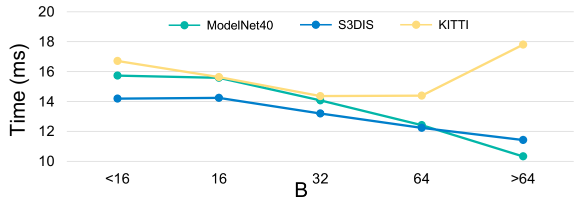


*Figure 14.* Execution time of different channel size threshold $B$.

However, this dataflow inevitably leads to redundant computation and incurs high data movement costs, resulting in significant performance degradation when the channel size increases.

## 9 Discussion

Considering that some GPUs have no tensor core unit (Markidis et al., 2018), and the usage of tensor cores leads to precision issues, optimization based on tensor cores is not included in this work. Nevertheless, tensor core support (Feng et al., 2021; Wang et al., 2023) can be combined with the proposed designs smoothly, which enables the performance to be further enhanced. E.g., in an FGMS operation, with the indicated mapping address, the input features and the weight matrix can be tiled and fed into tensor cores for matrix multiplication computation to achieve higher throughput. (Ye et al., 2023)

## 10 Conclusion

An efficient sparse convolution engine for 3D point clouds, PCEngine, is proposed in this paper. PCEngine innovatively proposes techniques like the coded-CSR format and the indicator-assisted segmented FGMS fusion scheme to improve utilization of GPU hardware for the sparse convolution operation. PCEngine subtly points out that input dynamics damage the performance of previous sparse convolution engines, and introduces the heuristic adaptive dataflow for the problem. Extensive experimental results show that PCEngine achieves $1.81\times$ and $1.64\times$ speedup on average for sparse convolution operation and end-to-end 3D point cloud networks, respectively. More importantly, we believe that the insights and techniques in this paper can benefit other sparse problems.

## References

Armeni, I., Sener, O., Zamir, A. R., Jiang, H., Brilakis, I., Fischer, M., and Savarese, S. 3D Semantic Parsing of Large-scale Indoor Spaces. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1534–1543, June 2016.

Boulch, A. Generalizing Discrete Convolutions for Unstructured Point Clouds. 2019.

Choy, C., Gwak, J., and Savarese, S. 4D Spatio-Temporal ConvNets: Minkowski Convolutional Neural Networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 3075–3084, 2019.

Deng, J., Shi, S., Li, P., Zhou, W., Zhang, Y., and Li, H. Voxel R-CNN: Towards High Performance Voxel-based 3D Object Detection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pp. 1201–1209, 2021.

Feng, B., Wang, Y., Chen, G., Zhang, W., Xie, Y., and Ding, Y. EGEMM-TC: Accelerating Scientific Computing on Tensor Cores with Extended Precision. In *Proceedings of 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 278–291, 2021.

Geiger, A., Lenz, P., and Urtasun, R. Are We Ready for Autonomous Driving? The KITTI Vision Benchmark Suite. In *2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 3354–3361, 2012.

Graham, B. Sparse 3d convolutional neural networks. *arXiv preprint arXiv:1505.02890*, 2015.

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

Graham, B., Engelcke, M., and Van Der Maaten, L. 3D Semantic Segmentation with Submanifold Sparse Convolutional Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 9224–9232, 2018.

He, K., Zhang, X., Ren, S., and Sun, J. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778, 2016.

Lin, Y., Zhang, Z., Tang, H., Wang, H., and Han, S. PointAcc: Efficient Point Cloud Accelerator. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 449–461, 2021.

Liu, Y., Fan, B., Xiang, S., and Pan, C. Relation-shape Convolutional Neural Network for Point Cloud Analysis. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 8895–8904, 2019.

Mao, J., Shi, S., Wang, X., and Li, H. 3D Object Detection for Autonomous Driving: A Review and New Outlooks. *arXiv preprint arXiv:2206.09474*, 2022.

Markidis, S., Der Chien, S. W., Laure, E., Peng, I. B., and Vetter, J. S. Nvidia tensor core programmability, performance & precision. In *2018 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, pp. 522–531. IEEE, 2018.

Maturana, D. and Scherer, S. VoxNet: A 3D Convolutional Neural Network for Real-time Object Recognition. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 922–928. IEEE, 2015.

Nvidia. CUTLASS. https://github.com/NVIDIA/cutlass, 2023.

Riegler, G., Osman Ulusoy, A., and Geiger, A. OctNet: Learning Deep 3D Representations at High Resolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 3577–3586, 2017.

Ronneberger, O., Fischer, P., and Brox, T. U-Net: Convolutional Networks for Biomedical Image Segmentation. In *International Conference on Medical Image Computing and Computer-assisted Intervention*, pp. 234–241. Springer, 2015.

Rusu, R. B. and Cousins, S. 3D is Here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 9-13 2011. IEEE.

Shi, S., Wang, Z., Shi, J., Wang, X., and Li, H. From Points to Parts: 3D Object Detection from Point Cloud with Part-aware and Part-aggregation Network. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 43 (8):2647–2664, 2020.

Tang, H., Liu, Z., Li, X., Lin, Y., and Han, S. TorchSparse: Efficient Point Cloud Inference Engine. In *Proceedings of Machine Learning and Systems*, pp. 302–315, 2022.

Traveller59. Spconv: Spatially Sparse Convolution Library. https://github.com/traveller59/spconv, 2022.

Wang, P.-S., Liu, Y., Guo, Y.-X., Sun, C.-Y., and Tong, X. O-CNN: Octree-based Convolutional Neural Networks for 3D Shape Analysis. *ACM Transactions On Graphics (TOG)*, 36(4):1–11, 2017.

Wang, X., Ang, M. H., and Lee, G. H. Voxel-based Network for Shape Completion by Leveraging Edge Generation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 13189–13198, October 2021.

Wang, Y., Sun, Y., Liu, Z., Sarma, S. E., Bronstein, M. M., and Solomon, J. M. Dynamic Graph CNN for Learning on Point Clouds. *Acm Transactions On Graphics (tog)*, 38(5):1–12, 2019.

Wang, Y., Feng, B., Wang, Z., and Ding, Y. TC-GNN: Accelerating Sparse Graph Neural Network Computation Via Dense Tensor Core on GPUs. *arXiv preprint arXiv:2112.02052*, 2023.

Wen, X., Li, T., Han, Z., and Liu, Y.-S. Point Cloud Completion by Skip-attention Network with Hierarchical Folding. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 1939–1948, 2020.

Wu, W., Qi, Z., and Fuxin, L. PointConv: Deep Convolutional Networks on 3D Point Clouds. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 9621–9630, 2019.

Wu, Z., Song, S., Khosla, A., Yu, F., Zhang, L., Tang, X., and Xiao, J. 3D ShapeNets: A Deep Representation for Volumetric Shapes. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1912–1920, June 2015.

Yan, Y., Mao, Y., and Li, B. Second: Sparsely Embedded Convolutional Detection. *Sensors*, 18(10):3337, 2018.

Ye, Z., Lai, R., Shao, J., Chen, T., and Ceze, L. Sparsetir: Composable abstractions for sparse compilation in deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pp. 660–678, 2023.

Yin, T., Zhou, X., and Krahenbuhl, P. Center-based 3D Object Detection and Tracking. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 11784–11793, 2021.

Zhai, Y., Jiang, C., Wang, L., Jia, X., Zhang, S., Chen, Z., Liu, X., and Zhu, Y. ByteTransformer: A High-Performance Transformer Boosted for Variable-Length Inputs. *arXiv preprint arXiv:2210.03052*, 2023.

Zhu, X., Zhou, H., Wang, T., Hong, F., Ma, Y., Li, W., Li, H., and Lin, D. Cylindrical and Asymmetrical 3D Convolution Networks for Lidar Segmentation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 9939–9948, 2021.

## A Strided Sparse Convolution

Usually, the submanifold sparse convolution restricts the outputs only to the set of inputs with non-zero features so that the data sparsity will not dilate as the network grows deeper, which can also be interpreted as a stride equal to one situation. In (Graham & van der Maaten, 2017), it also talks about settings like strides larger than one. Specifically, the input coordinates are quantified by the stride to downsample the input set and derive the output set. The application of the strided sparse convolutional layer enables data downsampling through networks.

## B Inverse Sparse Convolution

In order to restore the original spatial shape of the input data (e.g., in MinkUNet), the inverse sparse convolution is introduced into the sparse convolutional networks to do the upsampling work. A strided sparse convolutional layer shares the opposite input and output set with its inverse sparse convolutional layer so that the downsampled data can be upsampled into the original spatial shape. In order to do that, the mapping of the strided layer is stored and shared with its inverse layer for reuse. Because of that, we are able to find the layers using the same mapping based on the network structure in Section 6 for heuristic designs.

## C Mapping Search

We present the details of the mapping search operation here. Vanilla mapping search can be divided into three steps: (1) Output set generation. (2) Building input hash table. (3) Querying the hash table to derive the mapping. As the coded-CSR format mapping is utilized in our design, we need an extra step (4) to cover the format conversion and the encoding overhead, and step (4) is denoted as mapping processing.

In the mapping search operation, first, the output set can be directly generated based on the input set. Specifically, the output set is the same as the input set for a submanifold sparse convolutional layer, and the output set takes a subset of the input set in a quantization manner for a strided sparse convolutional layer. Then we build a hash table that maps the input coordinates to the input indices in $\mathcal{O}(1)$ time. Before querying the hash table, we calculate the candidate input coordinates based on the convolution dependency of inputs, outputs, and weights. A candidate coordinate is used in a query to the hash table, and if a valid input index can be reached, the corresponding dependency of the input, output, and weight becomes a valid mapping.

In general, we use the same paradigm for mapping search operations as the previous works. Besides, we design the whole process into a pipeline to reduce the mapping search latency. As is depicted in Figure 15, we decouple the step (4) mapping processing into two independent operators to separately deal with the input or output mapping, and schedule all the operators into two streams. The two streams execute operators in a parallel manner so that the overall latency can be reduced.

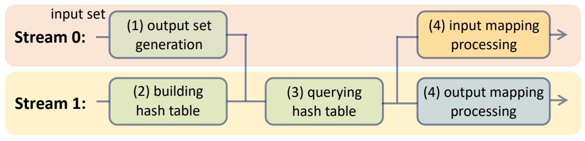

input set
Stream 0:
(1) output set generation
(4) input mapping processing
Stream 1:
(2) building hash table
(3) querying hash table
(4) output mapping processing

*Figure 15.* Pipeline design of the mapping search operation.

## D Artifact Appendix

### D.1 Abstract

This section is mainly the guideline to perform artifacts evaluation for the paper. The source code of PCEngine includes an optimized CUDA and C/C++ backend and a PyTorch-based frontend. The optimized sparse convolution kernel is accessed through the Python wrapper. The following context shows how to build and use the runtime library to conduct the evaluation.

### D.2 Artifact check-list (meta-information)

- **Algorithm:** Inference stage of sparse convolution on GPU and the optimizations.
- **Program:** CUDA, C/C++ and Python code.
- **Compilation:** nvcc11.1 with -O3 flag.
- **Run-time environment:** Ubuntu 20.04 with CUDA SDK 11.1 installed.
- **Hardware:** Any Nvidia GPUs with compute capability >= 7.0 (Recommended GPU: Nvidia GeForce RTX 3090 or Nvidia GeForce RTX 2080).
- **Expected memory required to run the artifact:** with 64GB main memory and 8GB GPU Memory.
- **Expected time to run the experiments:** 5 hours.
- **Public available:** Yes.

### D.3 Description

#### *D.3.1 How delivered*

The source code is available in the form of Github repository (`https://github.com/hkeee21/PCEngine`) and archived on Zenodo (`https://doi.org/10.5281/zenodo.7893091`).

#### *D.3.2 Hardware dependencies*

The implementation works on Intel x86 CPUs and Nvidia GPUs.

#### *D.3.3 Software dependencies*

- CUDA 11.1+
- PyTorch 1.10.0+
- TorchSparse 2.0.0+
- SpConv 2.2.3+

#### *D.3.4 Datasets*

- ModelNet40
- S3DIS
- KITTI

### D.4 Installation

The runtime library is built based on PyTorch C/C++ extensions. Please follow the `README.md` in the source code repository to install PCEngine.

### D.5 Experiment workflow

Before the artifacts evaluation can be conducted, please follow the `README.md` in the source code repository to install requirements and download datasets. For evaluation, we have prepared a Python script for each figure result.

### D.6 Evaluation and expected result

Please follow the `README.md` to run Python scripts, and the evaluation results will be generated into `.csv` files and stored in `evaluation/results/` directory.