TorchSparse++: Efficient Point Cloud Engine

Haotian Tang¹,∗ Shang Yang¹,²,∗ Zhijian Liu¹,∗
Ke Hong² Zhongming Yu⁴ Xiuwu Li⁵ Guohao Dai⁵ Yu Wang² Song Han¹
¹MIT ²Tsinghua University ³Shanghai Jiao Tong University ⁴UCSD ⁵UC Berkeley

https://torchsparse.mit.edu

Abstract

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

1. Introduction

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

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

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

∗ indicates equal contributions. Contact: kentang@mit.edu.
achieving state-of-the-art performance on real-world workloads. However, all these libraries assume a single dataflow throughout the execution of the entire model, which limits the design space for kernel optimization, such as tiling parameters.

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

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

2. Library

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

2.1. Overview

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

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

\[
x_{\text{out}}^k = \sum_{\delta \in \Delta^D(K)} \sum_j 1[p_j = sq_k + \delta] (x_{j}^{\text{in}} \cdot W_{\delta}),
\]

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

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

2.2. APIs and Integration

APIs. TorchSparse++ offers easy-to-use and PyTorch-like APIs, depicted in Figure 3 and Figure 4. Both module and functional implementations are provided for all sparse point cloud operators. To construct a sparse convolutional network for point clouds, only a conversion from PyTorch’s \nn.Conv3d, \nn.BatchNorm3d, and \nn.ReLU to our spnn counterpart is required, as shown in Figure 3. Unlike other libraries, users do not need to specify any additional fields at module initialization time in TorchSparse++. For example, SpConv requires an indice_key field during convolution class initialization to help the system exploit map reuse.
Figure 3. TorchSparse++ offers user-friendly, PyTorch-like interfaces that allow seamless support of both training and inference. It can also be easily integrated into open-source frameworks such as mmdet3d [13] with less than 10 lines of code.

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

Integration. It is easy to integrate TorchSparse++ into existing algorithm frameworks and open-source repositories. For example, as in Figure 3, frameworks like mmdet3d [13] often provide registries to support building modules from a configuration dictionary. Registering operators in TorchSparse++ is as simple as registering a Conv2d layer to these frameworks. Users can simply modify the type field in the configuration dictionary passed to build_conv_layer from Conv2d to SparseConv3d to switch between the two layer types. All other layer parameters, such as input and output channels, kernel size, stride, padding, and bias, retain their equivalent meanings to their PyTorch 2D counterpart. We provide an example of open-source integration of TorchSparse++ in https://github.com/mit-han-lab/bevfusion, where our system serves as the backend for the LiDAR backbone in a multi-modality 3D perception model [23]. It is also extremely simple to implement new 3D deep learning models from scratch using TorchSparse++. We provide another example, available at https://github.com/mit-han-lab/spvnas, which demonstrates how our system can be applied to a complex 3D semantic segmentation model [36]. This model features custom voxelization/devoxelization operators and supports neural architecture search. Other existing implementations such as MinkowskiEngine and SpConv v2 could not provide such flexibility to support this model.

3. Implementation

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

3.1. Gather-GEMM-Scatter Dataflow

Overview. A gather-GEMM-scatter [16, 40] implementation of sparse convolution (Figure 6) will finish all computation for one weight per before moving on to the other. It has an outmost host loop over kernel offsets. For each offset $\delta$, we first calculate all pairs $M_\delta = \{(p_j, q_k)\}$ such that $p_j = sq_k + \delta$. As is shown in Figure 5a, we then group all input features $\{f_j\}$ together, resulting in a
Figure 6. Illustration of the gather-GEMM-scatter dataflow for Figure 2 workload: we first gather input features according to $M_\delta$ for each weight $\delta$, then perform GEMM or batched GEMM, and finally scatter the results back to output locations given in $M_\delta$.

A $|M_\delta| \times C_{in}$ matrix in DRAM, and multiply it with weight $W_\delta \in \mathbb{R}^{C_{in} \times C_{out}}$, finally we scatter the results back to output positions $\{x_{out}^k\}$ according to $M_\delta$. For example, since $p_0 = 1 \times q_1 + (-1, -1)$, $p_1 = 1 \times q_5 + (-1, -1)$, we group features $x_{in}^k$, $x_{in}^q$ together, multiply them by $W_{-1,-1}$ and scatter back to output $x_{out}^1$, $x_{out}^5$.

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

3.2. Fetch-on-Demand Dataflow

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

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

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

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

3.3. Implicit GEMM Dataflow

Overview. A very recent advance in sparse convolution inference, SpConv v2 [39, 40], takes an alternative approach of implicit GEMM computation, which is originally designed for dense convolution on images in CUTLASS [18], cuDNN [11] and has been implemented in PointAcc [21], a specialized accelerator for point cloud workloads. As in Figure 7, the sparse convolution workload in Figure 2 is equivalent to a dense GEMM $C = A_{M \times K} \times B_{K \times N}$ with $M = N_{out}$ (number of output points), $N = C_{out}$, $K = |\Delta^D(K)| C_{in}$. We visualize matrix $A$ in Figure 7, where we have $N_{out} \times |\Delta^D(K)|$ grids and each grid corresponds to...
C\textsubscript{in} channels. Grids in green corresponds to input features. For example, output point \( q \) has four neighbors: \( p_0 \) (with \( w_{-1,-1} \)), \( p_1 \) (with \( w_{0,0} \)), \( p_2 \) (with \( w_{0,1} \)) and \( p_3 \) (with \( w_{1,0} \)).

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

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

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

4. Evaluation

4.1. Setup

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

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

4.2. Inference Speedup

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

5. Related Work

Point Cloud Inference Engines. Researchers have extensively developed efficient inference engines for sparse convolution. SpConv [40] proposes grid-based map search and the gather-GEMM-scatter dataflow. SparseConvNet [16] proposes hashmap-based map search and is later significantly improved (in latency) by MinkowskiEngine, which also introduces a new fetch-on-demand dataflow that excels at small workloads and allows generalized sparse convolution on >3D point clouds and on arbitrary coordinates.
TorchSparse\cite{35} pushes the performance of gather-GEMM-scatter by fusing memory operations together and adaptively grouping computation into batches. It also implements a GPU hash table and accelerates map search by exploiting kernel fusion and symmetry. More recently, SpConv v2\cite{40} switches to the sorted implicit GEMM workflow, inspired by CUTLASS\cite{18}. It entirely rewrites CUTLASS for sparse workload and achieves remarkable speedup; however, it is engineering-expensive and hard to maintain. All the existing dataflows are further optimized in this work.

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

**6. Conclusion**

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

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


