



TORCHSPARSE: EFFICIENT POINT CLOUD INFERENCE ENGINE

Haotian Tang^{*1} Zhijian Liu^{*1} Xiuyu Li^{*2} Yujun Lin¹ Song Han¹

<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, and 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 adopts *adaptive MM 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

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

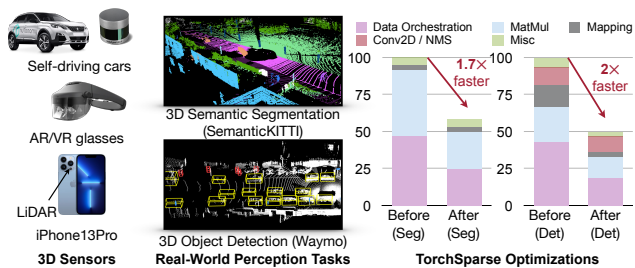


Figure 1: Widely available 3D sensors (left) have enabled more and more real-world AI applications to perceive the world using 3D point clouds (middle). However, processing 3D point cloud requires sparse computation that is not favored by general-purpose hardware. TorchSparse reduces the irregular computation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

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.

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,

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 1080 Ti 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. We first propose adaptive MM grouping to smartly batch the computation workloads from different kernel offsets together, trading #FLOPs for better regularity. We then 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 end-to-end 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

For a convolution of kernel size K , let $\mathbf{x}_j \in \mathbb{R}^C$ be a C -dimensional feature vector for the coordinates $\mathbf{p}_j \in \mathbb{Z}^D$ in a D -dimensional space, $\mathbf{W} \in \mathbb{R}^{K^D \times C^{\text{in}} \times C^{\text{out}}}$ be weights, and $\Delta^D(K)$ be the list of offsets in the convolution kernel. For instance, $\Delta^2(5) = \{-2, -1, 0, 1, 2\}^2$ and $\Delta^3(3) = \{-1, 0, 1\}^3$. The weights can be broken down into K^D ma-

trices of shape $C^{\text{in}} \times C^{\text{out}}$, denoted as \mathbf{W}_δ for $\delta \in \Delta^D(K)$. Then, convolution with stride s can be represented as,

$$\mathbf{x}_k^{\text{out}} = \sum_{\delta \in \Delta^D(K)} \mathbf{x}_j^{\text{in}} \cdot \mathbf{W}_\delta \quad \text{for } \mathbf{p}_j \in \mathbf{P}^{\text{in}}, \mathbf{q}_k \in \mathbf{P}^{\text{out}} \quad (1)$$

Here, $\mathbf{p}_j = s \cdot \mathbf{q}_k + \delta$. In dense convolution (Figure 3a), each nonzero input is multiplied with all nonzero weights and nonzeros quickly dilate ($\mathbf{P}^{\text{in}} \subset \mathbf{P}^{\text{out}}$). Contrarily, SparseConv (Figure 3b) only performs computation specified by *maps* ($(\mathbf{p}_j, \mathbf{q}_k, \mathbf{W}_\delta)$ in Equation 1, or alternatively written as $(\mathbf{p}_j, \mathbf{q}_k, \mathbf{W}_n)$, with n to be weight index) and keeps the input sparsity unchanged ($\mathbf{P}^{\text{in}} = \mathbf{P}^{\text{out}}$). SparseConv will iterate over all maps and perform $\mathbf{x}_k^{\text{out}} = \mathbf{x}_k^{\text{out}} + \mathbf{W}_\delta \cdot \mathbf{x}_j^{\text{in}}$.

2.1 Mapping Operations

Mapping is the procedure to find the input-output *maps* $\mathcal{M} = \{(\mathbf{p}_j, \mathbf{q}_k, \mathbf{W}_\delta)\}$ for sparse convolution. Here, j is the index of input point \mathbf{p} in \mathbf{P}^{in} , k is the index of output point \mathbf{q} in \mathbf{P}^{out} , and \mathbf{W}_δ is the weight matrix for offset δ . Generating maps typically requires two steps: calculating the output coordinates \mathbf{P}^{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.*, $\mathbf{P}^{\text{out}} = \mathbf{P}^{\text{in}}$.

When the convolution stride is larger than 1 (*i.e.*, downsampling), we refer the readers to Appendix A for the detailed output coordinates calculation algorithm. Basically, the nonzero inputs will first dilate (*i.e.*, $\mathbf{p} - \delta$), and only points on the strided grids within boundaries will become outputs \mathbf{q} , that is, $s \cdot \mathbf{q} = \mathbf{p} - \delta$. Take input coordinates (3, 5) as an example (with stride of 2). For kernel offset $\delta = (1, 1)$, output coordinates will be $((3, 5) - (1, 1)) / 2 = (1, 2)$; for kernel offset $\delta = (0, 0)$, there is no valid output since $((3, 5) - (0, 0))$ are not multiples of stride $s = 2$.

2.1.2 Map Search

As in Algorithm 1, sparse convolution will travel all possible input coordinates for each output coordinates to search maps. A map is generated only when the input is nonzero.

To efficiently examine whether the possible input $\mathbf{q}_j + \delta$ is nonzero, a common implementation is to record the coordinates of nonzero inputs with hash table. The key-value pairs are (key=input coordinates, value=input index), *i.e.*, (key = \mathbf{p}_j , value = j). The hash function can simply be flattening the coordinate of each dimension into an integer. Notice that for layers with odd kernel size and stride=1, the maps are symmetric (see Figure 2).

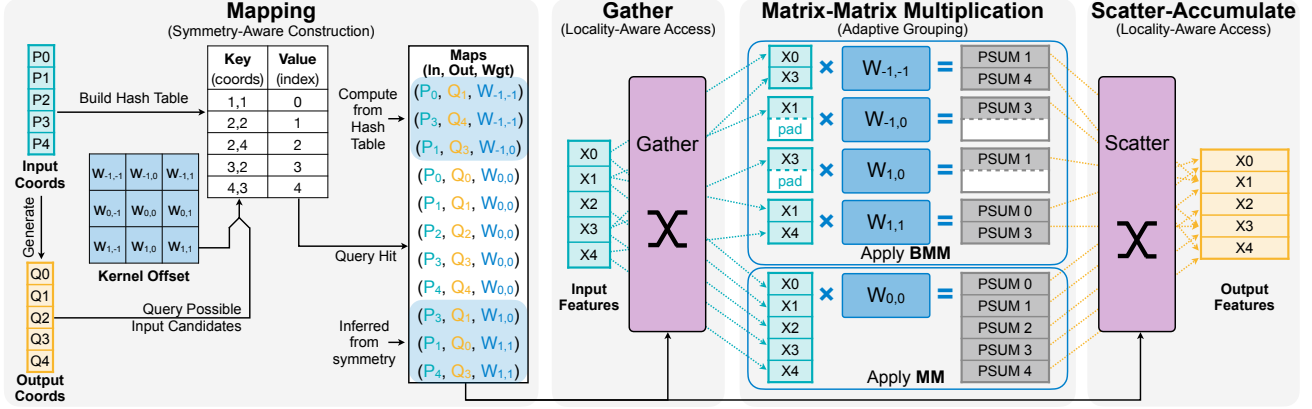


Figure 2: TorchSparse aims at accelerating *Sparse Convolution*, which consists of four stages: mapping, gather, matmul and scatter-accumulate. We follow two general principles that ① memory footprint should be reduced ② computation regularity should be increased to optimize these four components with quantized, vectorized, row-major scatter/gather (Principle ①); adaptively batched matmul (Principle ②) and mapping kernel fusion (Principle ①).

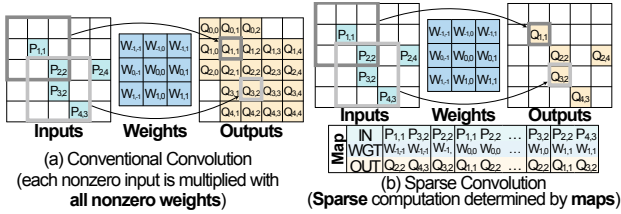


Figure 3: SparseConv (b) does *not* multiply each nonzero input with all nonzero weights as conventional convolution (a) does.

2.2 Data Orchestration and Matrix Multiplication

After maps are generated, sparse convolution will multiply the input feature vector x_j^{in} with corresponding weight matrix W_δ and accumulate to the corresponding output feature vector x_k^{out} , following the map (p_j, q_k, W_δ) .

Since the utilization ratio of matrix-vector multiplication is low on GPU, a better implementation will follow Gather-MatMul-Scatter computation flow as shown 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 P^{in} , output coordinates P^{out}
kernel size K , stride s

Output: maps \mathcal{M}

```

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

Algorithm 2 Gather-MatMul-Scatter

Input: input features X^{in} , weights W , maps \mathcal{M}

Output: output features X^{out}

```

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

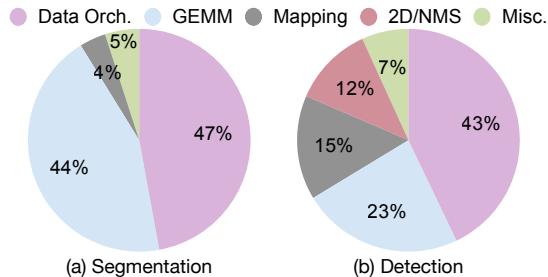


Figure 4: Data Movement and Mapping Operations constitute a significant proportion of SparseConv runtime.

dilate during the inference, *i.e.*, $P^{\text{in}} \subset P^{\text{out}}$ (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 $P^{\text{in}} = P^{\text{out}}$, 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 have to search maps for **every** down-sampling block. Furthermore, graph convolution shares same weight matrix for different neighbors, *i.e.* all W_δ 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 δ 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 SparseConvNets: one for segmentation (Figure 4a) and one for detection (Figure 4b). Based on observations in Figure 4, we summarize two principles for SparseConv 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 SparseConv and takes up a large proportion of total execution time (20%-50%). Despite the fact that Algorithm 2 decouples

actual matmul computation from data movement and well-optimized libraries such as cuDNN can be directly used to calculate $X^{\text{out}} \leftarrow X^{\text{in}} \cdot W_\delta$, it turns out that actual matmul 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*). Consequently, matmul in MinkUNet ($0.5 \times$ width) runs at 8.1 TFLOP/s on RTX 2080 Ti with FP16 quantization, which means only 30.1% device utilization. As such, *improving the regularity* in matmul computation will potentially be helpful: we boost the utilization to 44.2% after optimization (detailed in Table 2).

Principle II. Reduce Memory Footprint On average, data movement is the largest bottleneck in SparseConvNets, which takes up 40%-50% of total runtime. This is because scatter-gather operations are bottlenecked by GPU memory bandwidth (*limited*) instead of computation resources (*abundant*). Worse still, the dataflow in Algorithm 2 completely separates scatter-gather operations for different weights. This further ruins the possibility of potential reuse in data movement, which will be detailed in Figure 9. It is also noteworthy that the large mapping latency in the Center-Point detector also stems from memory overhead: typical hashmap implementations for Algorithm 1 require multiple DRAM accesses on a hash collision, and a naïve implementation of output coordinate calculation (see Appendix A) also requires multiple DRAM writes for intermediate results. Consequently, *reducing memory footprint* is at the heart of data movement and mapping optimization, which is described in Section 4.

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). Section 4.4 analyzes the opportunities to speedup 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

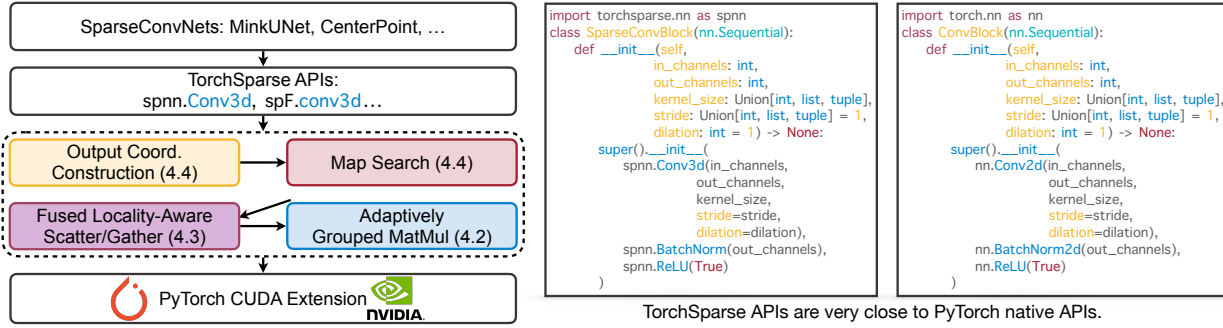


Figure 5: System Overview for TorchSparse: TorchSparse provides handy Python APIs similar to PyTorch and applies low-level optimizations to data movement, matrix multiplication and mapping operations in SparseConv.

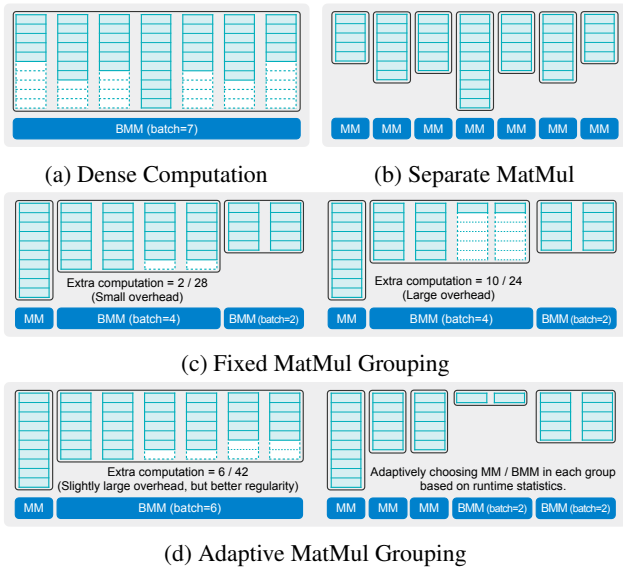


Figure 6: Illustration for different MatMul grouping strategies in TorchSparse (c, d) comparing with baselines (a: large FLOPs overhead, b: low device utilization, many kernel calls). Fixed grouping (c) trades FLOPs for regularity and adaptive grouping (d) searches for the best balance point.

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.

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 matmul operations 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 matmul for different weights together. We find it helpful to introduce

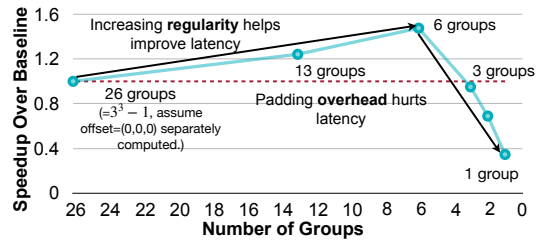


Figure 7: Trading FLOPs for computation regularity via batched matrix multiplication can potentially bring about 1.5x speedup in matmul.

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 matmul computation 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 matmul computation.

4.2.1 Symmetric MatMul Grouping

With sparse workload, the map sizes for different weights within one sparse convolution layer are usually *different*, as opposed to the dense workload (Figure 6a). However, for *submanifold* (stride=1) layers with odd kernel sizes, it turns out that the maps corresponding to weight offset (a, b, c) will always have the same size as the maps corresponding to the *symmetric* weight offset $(-a, -b, -c)$. For a map $(p_j, q_k, W_{a,b,c})$, we have $q_k = p_j + (a, b, c)$. Then, $p_j = q_k + (-a, -b, -c)$, which implies that $(q_k, p_j, W_{-a,-b,-c})$ is also a valid map. As such, we can establish a one-to-one correspondence between maps for weights $\pm(a, b, c)$. Therefore, we are able to group matmul workload for symmetric weight offsets together and naturally have a batch size of 2. Note that the matmul workload corresponding to

weight offset $(0, 0, 0)$ is processed separately since it does not require any explicit data movement. As indicated in Figure 7, the batch=2 matmul (13 groups) can already be up to $1.2\times$ faster than separate matmul.

4.2.2 Fixed MatMul 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 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 W_0 to 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 (symmetric grouping is also enabled whenever possible). 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 MatMul 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.

Concretely, the adaptive grouping algorithm defines two auto-tuned parameters ϵ and S , where ϵ indicates our tolerance of redundant computation, and S is the threshold of the workload size, above which we should perform `mm` within each group, otherwise `bmm`. Given ϵ , we scan over sizes of all maps in the current workload for W_0 to 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 ϵ . 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. We further present the details of this algorithm in Appendix B. Note that even if ϵ and S are fixed, the generated strategy itself is still *input-adaptive*. Since different input point clouds have different map sizes, even

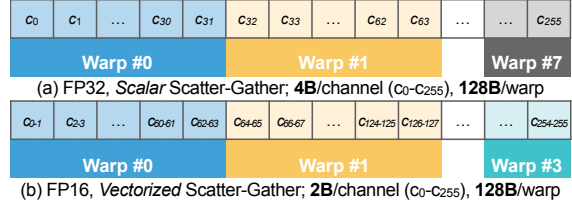


Figure 8: TorchSparse applies *vectorized* and *quantized* scatter-gather to greatly data movement latency.

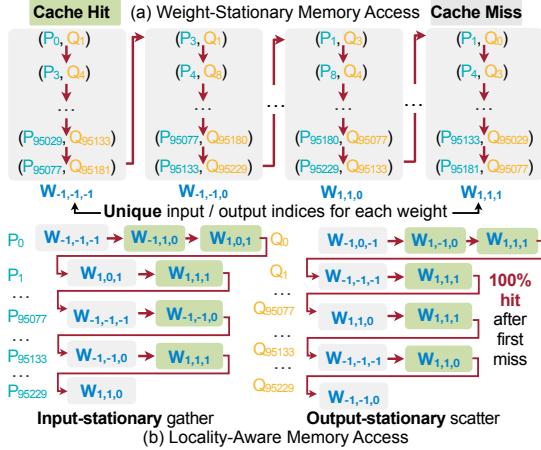


Figure 9: TorchSparse proposes cache-friendly *locality-aware* and memory access pattern. In contrary, baseline implementation (a) cannot exploit cache reuse due to uniqueness in input/output indices for each weight.

the same ϵ can potentially generate different group partition strategies for different samples. The (ϵ, 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 special cases.

For a given sparse CNN, we determine (ϵ, 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 and explores a space of $<1,000$ configurations, corresponding to <10 minutes of search time on desktop GPUs. The strategy derived on the small subset can be directly applied and does not require any parameter optimization during 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 data reuse.

4.3.1 Quantized and Vectorized Memory Access

Naturally, FP16 quantization brings $2\times$ theoretical DRAM access saving comparing with the FP32 baseline, but such reduction cannot be transformed to real speedup without vectorized scatter-gather, as in Figure 8.

Concretely, 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 ($1.3\times$) compared to the theoretical value ($2\times$) 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 $1.9\times$ 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 $N_1 = 2|\mathcal{M}|(C_{in} + C_{out})$, where \mathcal{M} is the map for this layer (defined in Section 2.1), and C_{in} and C_{out} correspond to input and output channel numbers. However, the total feature size of this layer is $N_2 = N_{in}C_{in} + N_{out}C_{out}$. Given the fact that usually $|\mathcal{M}|/N_{in} > 4$ on real-world point clouds, it means that the feature of each point is repetitively accessed for at least 4 times. As such, we can ideally have $1.6\times$ further DRAM access saving for scatter/gather (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 \mathbf{W}_{k+1} , the GPU cache is filled with *scatter* buffer features for \mathbf{W}_k as

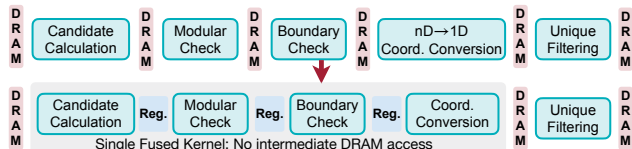


Figure 10: TorchSparse reduces mapping DRAM access and improves mapping latency via kernel fusion.

long as the GPU cache size is much smaller than N_1 (typically $> 40\text{MB}$, much larger than the 5.5MB L2 cache of NVIDIA RTX 2080 Ti). Intuitively, for gather operation on \mathbf{W}_{k+1} , we hope that the cache is filled with gather buffer features from \mathbf{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.

Moreover, the order of memory access matters. In the weight-stationary / map order (Figure 9a), all map entries for weight \mathbf{W}_k are *unique*, so there is *no* chance of feature reuse and each gather/scatter will be a cache miss. As is shown 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. Specifically, for the i^{th} entry $(\mathbf{p}_j, \mathbf{q}_k, \mathbf{W}_n)$ in the map for \mathbf{W}_n , we maintain a input neighbor list for the input point \mathbf{p}_j and add the entry (\mathbf{W}_n, i) into the list. During gathering, we traverse each point \mathbf{p}_j in order and for every (\mathbf{W}_n, i) in the input neighbor list, we write \mathbf{X}_j^{in} to DRAM location $\sum_{t=0}^{n-1} |\mathcal{M}[\mathbf{W}_t]| + i$. Note that \mathbf{X}_j^{in} is read from DRAM only once and held in the register. Hence, we achieve the maximal reuse for gathering. Similar technique is applied to scattering as well, where we read neighbor partial sums for each output point from DRAM, perform reduction in the register and write out the result only once. Experiments on real-world point cloud datasets for autonomous driving show up to $1.3\text{-}1.4\times$ speedup in data movement.

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 also reduce mapping overhead in SparseConvNet.

We first choose the map search strategy for each layer from [grid, hashmap] in a similar manner to adaptive grouping. Here, grid corresponds to a naive collision-free hashmap described in Section 2.1.2; it takes larger memory space but hashmap construction / query requires exactly 1 DRAM access per-entry, 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 and ①

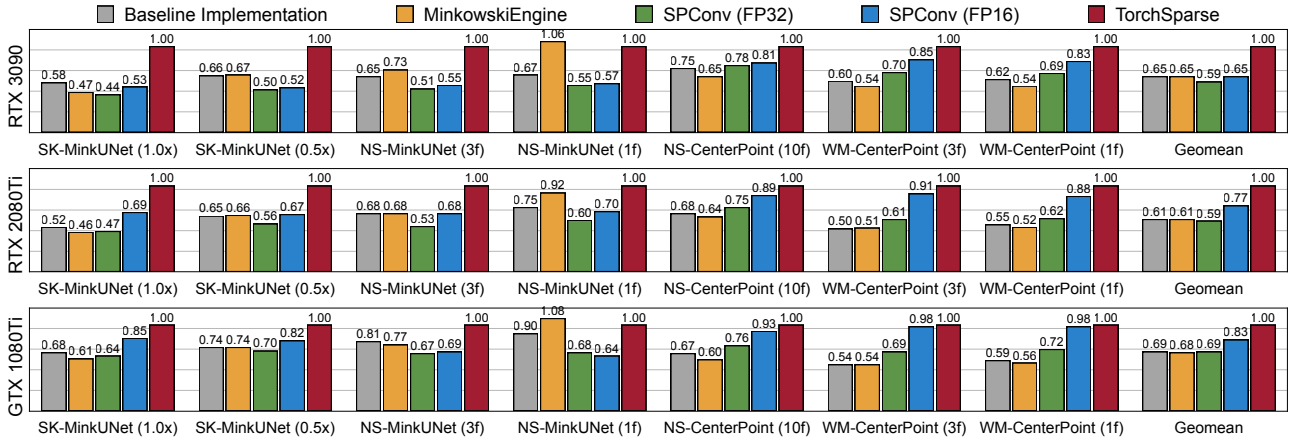


Figure 11: TorchSparse consistently outperforms state-of-the-art inference engines in both detection and segmentation benchmarks and achieves up to 1.5-1.6x geomean speedup, 2.3x single model speedup over MinkowskiEngine and SpConv.

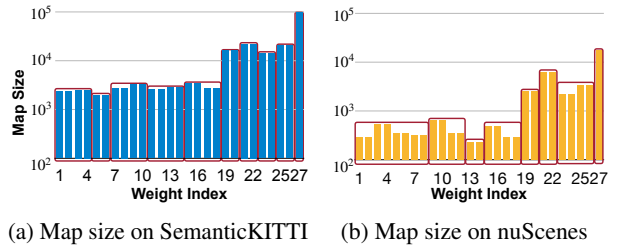
calculate candidate activated points with broadcast_add; ② perform modular check; ③ perform boundary check and generate a mask on whether each point is kept; ④ converts the remaining candidate point coordinates to 1D values; ⑤ perform unique operation to keep final output coordinates (detailed in Appendix A). There are DRAM accesses between every two of the five stages, making downsampling kernels memory-bounded. We therefore fuse stages ① to ④ in 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 4.6x on detection tasks after our optimization.

5 EVALUATION

5.1 Setup

TorchSparse is implemented in CUDA and we provide easy-to-use PyTorch-like interfaces (described in Section 4.1). We build TorchSparse based on PyTorch 1.9.1, CUDA 10.2/11.1 + 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 SparseConv inference engines MinkowskiEngine v0.5.4 (Choy et al., 2019) and SpConv v1.2.1 (Yan et al., 2018) on three generations of NVIDIA GPUs: GTX1080Ti, RTX2080Ti and RTX3090. 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 7 top-performing SparseConvNets on large-scale datasets: MinkUNet (Choy et al., 2019) (0.5x/1x width) on SemanticKITTI (Behley et al., 2019), MinkUNet (1/3 frames) on nuScenes-LiDARSeg (Caesar



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

Figure 12: Maps on nuScenes are much smaller than on SemanticKITTI for MinkUNet model. Thus, grouping strategy on NuScenes is more aggressive (8 groups vs. 10 groups).

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

We compare our TorchSparse with the baseline design, MinkowskiEngine and SpConv in Figure 11. In general, TorchSparse achieves the best performance comparing with other systems on RTX3090. Specifically, TorchSparse achieves up to 2.16x speedup on segmentation models and 1.6-2x speedup on detection models over MinkowskiEngine. We achieve smaller speedup on 1-frame MinkUNet model for the nuScenes dataset because MinkowskiEngine applies specialized optimizations to small models by using *fetch-on-demand* dataflow rather than gather-matmul-scatter dataflow shown in Algorithm 2.

TorchSparse also shows 1.2x faster inference compared with 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,

which will be discussed in Section 6.2. Note that we report *end-to-end* speedup in Figure 11 and 10% of total runtime in CenterPoint (Yin et al., 2021) are not related to point cloud computation (image convolution and non-maximum suppression, as is shown in Figure 4), therefore, the speedup ratio on *SparseConv* part of CenterPoint is 10% more. 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. Notice that segmentation models usually have smaller workload comparing with detectors, and thus the GPUs are usually more under-utilized, making it necessary to apply batching strategies to improve device utilization.

We also notice that TorchSparse achieves consistent speedup over other systems on GTX1080Ti, which has *no* FP16 tensor cores. However, comparison with the baseline design clearly shows that we still achieve $1.5\times$ geomean speedup, only 11% less than the speedup we achieved on 2080Ti, which is equipped with tensor cores. This indicates that the native tensor-core speedup on tensor-core-enabled GPUs only constitutes a very minor proportion of our performance gain comparing with the optimizations in Section 4.

TorchSparse runs at 36, 26, 13 FPS for MinkUNet ($1.0\times$) on SemanticKITTI on RTX3090, RTX2080Ti, GTX1080Ti, respectively, all satisfying real time requirement (≥ 10 FPS). For the 3-frame model on nuScenes-LiDARSeg, TorchSparse provides 45, 40, 25 FPS throughput on three devices, at least $2\times$ faster than the LiDAR frequency. Even for the heaviest 3-frame CenterPoint model on Waymo, the TorchSparse backend is still able to run it at 10.3 FPS on the GTX1080Ti GPU. As such, our system paves the way for real-time 3D perception on autonomous driving cars.

6 ABLATION STUDY

6.1 Effect of Matrix Multiplication Optimization

We first examine the performance of different grouping strategies Figure 6 on SemanticKITTI with MinkUNet ($0.5\times$) and on nuScenes with MinkUNet (3 frames). In general, our adaptive grouping strategy outperforms all handcrafted, fixed strategy and achieves $1.4\text{-}1.5\times$ over no grouping baseline. The results in Table 2 also suggests manually-designed strategy usually cannot generalize to all datasets: fixed 3-batch grouping achieves large speedup ($1.5\times$) 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 ϵ in our adaptive grouping algorithm. We also show the effectiveness

Specialization for Different Datasets		Optimized for	
		SemanticKITTI	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 ($1.0\times$)	MinkUNet ($0.5\times$)
Execute on	MinkUNet ($1.0\times$)	10.11	10.70
	MinkUNet ($0.5\times$)	5.37	4.72

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

Specialization for Different Hardware		Optimized for	
		RTX2080Ti	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 ($1.00\times$)	10.4 TFLOP/s ($1.00\times$)
Symmetric	8.2 TFLOP/s ($1.02\times$)	14.6 TFLOP/s ($1.39\times$)
Fixed	8.7 TFLOP/s ($0.87\times$)	21.1 TFLOP/s ($1.50\times$)
Adaptive	11.9 TFLOP/s ($1.39\times$)	16.9 TFLOP/s ($1.54\times$)

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

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.

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: Ablation analysis on data movement: feature quantization, vectorized memory access, fused and locality-aware access brings about 1.3 \times , 1.5 \times , 1.4 \times speedup. Here G means gather and S means scatter.

6.2 Effect of Data Movement Optimization

We then perform ablation analysis on MinkUNet (Choy et al., 2019) (1.0 \times) on the SemanticKITTI dataset (Behley et al., 2019). As is shown in Table 3, naively quantizing features to 16-bit won’t provide significant speedup for scatter-gather: especially for gathering, the speedup ratio is only 1.17 \times , far less than the theoretical value (2 \times). Instead, quantized and *vectorized* scatter/gather improves the latency of scatter-gather by 1.93 \times , 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 \times** speedup on gathering, **2.61 \times** speedup on scattering and **2.72 \times** 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 Effect of Mapping Optimization

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 \times faster than a general hashmap-based solution thanks to its no-collision property, resulting in a 1.6 \times end-to-end speedup for mapping. Fusing four small kernels accelerates output construction by 2.1 \times and brings 1.5 \times 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 \times and pushes the final end-to-end mapping speedup to **4.6 \times** .

7 RELATED WORK

Deep Learning on Point Clouds. Early research (Chang et al., 2015; Qi et al., 2016; Cicek et al., 2016) first convert point clouds to dense volumetric representation and apply dense CNNs to extract features. Another stream of research (Qi et al., 2017a;b; Li et al., 2018; Wu et al., 2019; Thomas et al., 2019; Wang et al., 2019b) directly perform convolution on the k -nearest neighbor/spherical nearest neighbor of

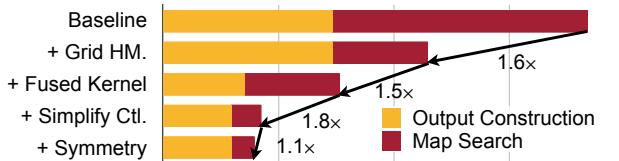


Figure 13: Ablation on mapping: grid-based hashmap, fused kernel, simplified control logic and utilizing symmetry brings about 1.6 \times , 1.5 \times , 1.8 \times and 1.1 \times speedup.

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 SparseConv, 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 SparseConv 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 SparseConv on >3D point clouds and on arbitrary coordinates.

8 CONCLUSION

We present TorchSparse, an efficient inference engine for emerging applications on point cloud perception. Guided by two general principles that we should trade computation for regularity and reduce memory footprint, we optimize matrix multiplication, data movement and mapping operations in SparseConv, achieving up to 1.5 \times , 2.7 \times and 4.6 \times speedup on these three components, resulting in up to 1.5-1.6 \times 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 SparseConv 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, Hyundai and Ford. 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. 1st 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.
- 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 (PAMI)*, 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., Kretschmar, H., Dotiwala, 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.

TORCHSPARSE: EFFICIENT POINT CLOUD INFERENCE ENGINE APPENDIX

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 P^{out} . We finally filter out duplicate coordinates in P^{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
   $P^{\text{out}} \leftarrow P^{\text{in}}$ 
else
   $P^{\text{out}} \leftarrow \emptyset$ 
  for  $p$  in  $P^{\text{in}}$  do
    # Traverse the neighbors of an input point.
    for  $\delta$  in  $\Delta^D(K)$  do
      # Calculate the candidate coordinates.
       $u \leftarrow p - \delta$ 
      # Add output if it passes modular and boundary check.
      if  $u \% s == 0$  and  $u < s \cdot b$  then
         $P^{\text{out}} \leftarrow P^{\text{out}} \cup \{u/s\}$ 
      end if
    end for
  end for
  # Filter out duplicate coordinates.
   $P^{\text{out}} \leftarrow \text{Unique}(P^{\text{out}})$ 
end if

```

B ADAPTIVE GROUPING ALGORITHM

Here we provided detailed illustration for the adaptive grouping algorithm. The algorithm is divided into two parts: grouped matmul (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 ϵ , we return the working group to the groups list and move pointers to start a new group. Sec-

ond, for each group, we determine if batched matmul is performed on it based on the value of S .

Algorithm 4 Grouped Matrix Multiplication

Input: input features X^{in} , weights W , maps \mathcal{M} , redundant computation tolerance ϵ , mm/bmm threshold S

Output: output features X^{out}

```

 $G \leftarrow \emptyset$ 
 $i \leftarrow 0$ 
# Traverse each weight index with unique number of inputs.
while  $i < \text{range}(\lfloor \Delta^D(K).size/2 \rfloor)$  do
   $n_{\min} \leftarrow 0$ ;  $n_{\max} \leftarrow \text{len}(\mathcal{M}[W_i])$ 
  # Always push the first index to the current group.
   $g \leftarrow \{i\}$ ;  $i \leftarrow i + 1$ 
  for  $j$  in  $\text{range}(i, \lfloor \Delta^D(K).size/2 \rfloor)$  do
     $n \leftarrow \text{len}(\mathcal{M}[W_j])$ 
     $n_{\min} \leftarrow \min\{n, n_{\min}\}$ ,  $n_{\max} \leftarrow \max\{n, n_{\max}\}$ 
    # Push the index to the group if the ratio no larger than  $\epsilon$ .
    if  $1 - n_{\min}/n_{\max} \leq \epsilon$  then
       $g \leftarrow g \cup \{j\}$ 
    else
      # Otherwise return and start a new group.
      break
    end if
  end for
  # Push the returned group to the groups list.
   $G \leftarrow G \cup \{g\}$ 
end while
for  $g$  in  $G$  do
   $n_{\max} \leftarrow \max\{\text{len}(\mathcal{M}[W_i]) \text{ for } i \text{ in } g\}$ 
  # Pad inputs and apply bmm when workload smaller than  $S$ .
  if  $n_{\max} < S$  then
    gather input feature matrices  $F_i$  with  $X^{\text{in}}$  and  $\mathcal{M}[W_i]$ 
    for  $i \in g$ ; pad zeros to each  $F_i$  to become length  $n_{\max}$ ;
    perform batched matrix multiplication between  $F_{i \in g}$  and
     $W_{i \in g}$  and then scatter results to corresponding  $X^{\text{out}}$ 
  else
    # Otherwise apply mm.
    perform Algorithm 2 in main paper with  $X_i^{\text{in}}$  for  $i \in g$  to
    get  $X^{\text{out}}$ 
  end if
end for

```

B.2 Adaptive Strategy Search

For each layer, we search for a specific configuration to conduct adaptive grouping (i.e. auto-tune ϵ and S). The tuning algorithm is shown in Algorithm 5, where we enumerate ϵ, S in a predefined search space (usually < 1000 configurations), use Algorithm 4 to perform matrix multiplication for the target layer, and select the ϵ, S pair which leads to the smallest average latency.

TorchSparse: Efficient Point Cloud Inference Engine

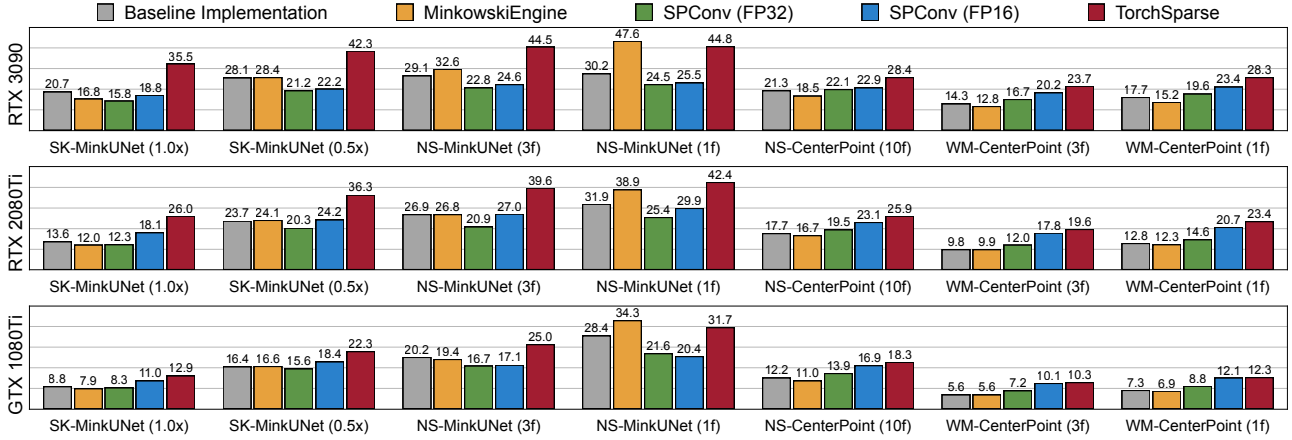


Figure 14: TorchSparse evaluation results in *absolute* values.

C RESULTS DETAIL

Algorithm 5 Adaptive Group Search

Input: sampled inputs subset D , weights W , maps \mathcal{M} ,
 redundant computation tolerance search space S_a ,
 mm/bmm threshold search space S_b

Output: selected redundant computation tolerance ϵ^* ,
 mm/bmm threshold S^*

```

 $f \leftarrow$  cost function to compute elapsed time on hardware
 $c_{\min} \leftarrow 0$ 
for  $\epsilon$  in  $S_a$  do
  for  $S$  in  $S_b$  do
     $c \leftarrow 0$ 
    for  $X^{\text{in}}$  in  $D$  do
       $c \leftarrow c + f(\text{run Algorithm 4 with } X^{\text{in}}, W, \mathcal{M}, \epsilon, S)$ 
    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
    
```

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.