TorchSparse++: Efficient Training and Inference Framework for Sparse Convolution on GPUs

Haotian Tang[*]
MIT
Cambridge, MA, USA
kentang@mit.edu

Shang Yang[*]
MIT, Tsinghua University
Cambridge, MA, USA
shangy@mit.edu

Zhijian Liu
MIT
Cambridge, MA, USA
zhijian@mit.edu

Ke Hong
Tsinghua University
Beijing, China
hjkl1379991@126.com

Zhongming Yu
UCSD
San Diego, CA, USA
zhy025@ucsd.edu

Xiuyu Li
UC Berkeley
Berkeley, CA, USA
xiuyu@berkeley.edu

Guohao Dai
Shanghai Jiao Tong University
Shanghai, China
daiguohao@sjtu.edu.cn

Yu Wang
Tsinghua University
Beijing, China
yu-wang@tsinghua.edu.cn

Song Han
MIT, NVIDIA
Cambridge, MA, USA
songhan@mit.edu

ABSTRACT

Sparse convolution plays a pivotal role in emerging workloads, including point cloud processing in AR/VR, autonomous driving, and graph understanding in recommendation systems. Since the computation pattern is sparse and irregular, specialized high-performance kernels are required. Existing GPU libraries offer two dataflow types for sparse convolution. The gather-GEMM-scatter dataflow is easy to implement but not optimal in performance, while the dataflows with overlapped computation and memory access (*e.g.* implicit GEMM) are highly performant but have very high engineering costs. In this paper, we introduce **TorchSparse++**, a new GPU library that achieves the best of both worlds. We create a highly efficient Sparse Kernel Generator that generates performant sparse convolution kernels at less than one-tenth of the engineering cost of the current state-of-the-art system. On top of this, we design the Sparse Autotuner, which extends the design space of existing sparse convolution libraries and searches for the best dataflow configurations for training and inference workloads. Consequently, TorchSparse++ achieves **2.9**×, **3.3**×, **2.2**× and **1.7**× measured end-to-end speedup on an NVIDIA A100 GPU over state-of-the-art MinkowskiEngine, SpConv 1.2, TorchSparse and SpConv v2 in inference; and is **1.2-1.3**× faster than SpConv v2 in mixed precision training across seven representative autonomous driving benchmarks. It also seamlessly supports graph convolutions, achieving **2.6-7.6**× faster inference speed compared with state-of-the-art graph deep learning libraries. Our code is publicly released at https://github.com/mit-han-lab/torchsparse.

[*] Both authors contributed equally to this research.

MICRO '23, October 28–November 01, 2023, Toronto, ON, Canada
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0329-4/23/10.
https://doi.org/10.1145/3613424.3614303

CCS CONCEPTS

• **Computer systems organization** → **Neural networks**.

KEYWORDS

GPU, neural network, sparse convolution, high-performance computing, point cloud, graph

ACM Reference Format:
Haotian Tang, Shang Yang, Zhijian Liu, Ke Hong, Zhongming Yu, Xiuyu Li, Guohao Dai, Yu Wang, and Song Han. 2023. TorchSparse++: Efficient Training and Inference Framework for Sparse Convolution on GPUs. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28–November 01, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3613424.3614303

1 INTRODUCTION

Sparse convolution [12, 18] plays a crucial role in a variety of cutting-edge applications, including augmented/virtual reality (AR/VR), autonomous driving, and recommendation systems. For instance, in advanced driver assistance systems (ADAS) and autonomous driving technology, data is collected from 3D sensors in the form of 3D point clouds. These point clouds often exhibit an exceptionally high spatial sparsity, with up to 99.99% spatial sparsity. In such cases, employing dense 3D convolutions for point cloud processing becomes inefficient. Likewise, social media graphs, like those found on platforms such as Twitter, exhibit even greater sparsity. As an illustration, the adjacency matrix of Twitter's social graph contains only a minuscule fraction, approximately 0.000214%, of the possible connections [56]. Therefore, there is a urgent need for efficient inference and training system for these sparse workloads.

Sparse convolution modifies the definition of regular convolution by only performing computation at a sparse set of output locations rather than the entire feature map. It is arguably the most important building block for almost all state-of-the-art 3D perception models (*e.g.* 3D semantic segmentation [10, 31, 41], 3D object detection [1, 6, 8, 17, 50, 51, 53, 58], 3D reconstruction [9],

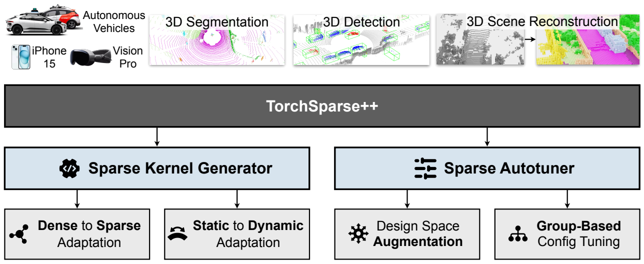


Figure 1: TorchSparse++ is a high-performance GPU library that provides highly-optimized dataflows for convolution on point clouds through a Sparse Kernel Generator. It further selects the optimal dataflow for each layer using the Sparse Autotuner, delivering up to 1.3-1.7× geomean inference and training speedups than the previous state-of-the-art system. Figures source: [2, 9, 34], Waymo, Cruise, Apple.

multi-sensor fusion [7, 27, 30], end-to-end navigation [29]). It also exhibits similar computation pattern to (relational) graph convolutions [19, 36]. Despite achieving dominant performance, the sparse and irregular nature of sparse convolution makes it harder to be processed on GPUs and there is no vendor library support. Dedicated libraries [18, 21, 40, 49, 50] with specialized high-performance kernels or even specialized hardware accelerators [14, 15, 28] are required for sparse convolution. As a result, many industrial driving assistance solutions still prefer pillar-based models [25], which flatten LiDAR points onto the BEV space and process them with a 2D CNN. These approaches cannot take full advantage of 3D geometry from LiDAR and tend to have much worse accuracy.

Several pioneering implementations of sparse convolution have adopted different dataflows for this operator. For instance, SparseConvNet [18] and SpConv v1 [50] use the vanilla gather-GEMM-scatter dataflow. It was improved by TorchSparse [40] that optimizes the gather-scatter paradigm through fusing memory operations and grouping computations adaptively into batches to improve device utilization. Dataflows based on gather-scatter can be implemented using vendor libraries with relative ease. However, they are fundamentally restricted in performance due to the inability to overlap memory access and computation. MinkowskiEngine [12] proposes the fetch-on-demand dataflow, which is optimized by PCEngine [21]. Recently, SpConv v2 [49, 50] has adapted the implicit GEMM dataflow for dense convolution to the sparse domain, achieving state-of-the-art performance on real-world workloads. Nevertheless, the best representative of these memory-computation overlapped dataflows, implicit GEMM, is extremely hard to implement. The metaprogrammer for SpConv v2 has more than 40k lines of code, making it hard for the community to further improve upon it.

To address the significant challenge of achieving both ease of implementation and state-of-the-art performance, we present *TorchSparse++* (Figure 1), a high-performance GPU library that combines the best of both worlds through the *Sparse Kernel Generator* and the *Sparse Autotuner*. Tackling a fundamentally *sparse* and *dynamic* workload, we propose a general method to adapt existing tensor compilers that are optimized for *dense* and *static* workloads, unlocking their potential to generate kernels that can deal with sparsity and variable workload shapes. On top of the generated kernels, we further extend the design space of existing point cloud libraries. We design a Sparse Autotuner to efficiently search for the best dataflow configurations through group-based tuning for a diverse set of workloads within the enlarged design space. The results of our Sparse Autotuner challenged the conventional design wisdom of using amount of computation, DRAM access or even total runtime for computation kernels as the indicator for end-to-end performance.

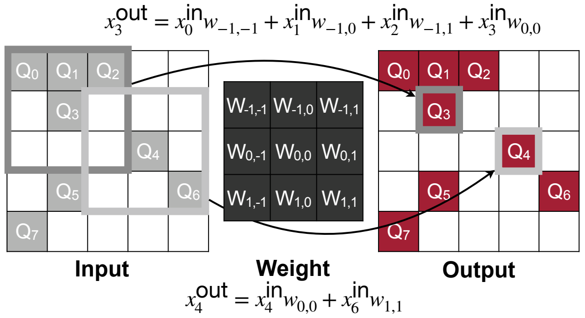

$x_3^{\text{out}} = x_0^{\text{in}} w_{-1,-1} + x_1^{\text{in}} w_{-1,0} + x_2^{\text{in}} w_{-1,1} + x_3^{\text{in}} w_{0,0}$
Q0
Q1
Q2
Q3
Q4
Q5
Q6
Q7
W-1,-1
W-1,0
W-1,1
W0,-1
W0,0
W0,1
W1,-1
W1,0
W1,1
Q0
Q1
Q2
Q3
Q4
Q5
Q6
Q7
Input
Weight
Output
$x_4^{\text{out}} = x_4^{\text{in}} w_{0,0} + x_6^{\text{in}} w_{1,1}$

Figure 2: Sparse convolution (Equation 1) on $\Delta^2(3)$: computation is performed only on *nonzero* inputs.

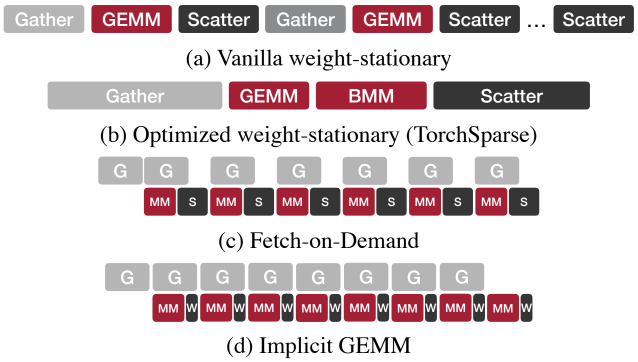
As a result, evaluated on seven representative models across three benchmark datasets, TorchSparse++ achieves end-to-end speedups of **2.9**×, **3.3**×, **2.2**×, and **1.7**× on an NVIDIA A100 GPU, at least **1.4**×, **1.8**×, **2.4**×, **2.2**× speedup on three other cloud GPUs from Pascal to Ampere architectures in three data precisions over state-of-the-art systems such as MinkowskiEngine, SpConv 1.2, TorchSparse, and SpConv v2. Additionally, in the training phase, our system outperforms the best existing training system, SpConv v2, by **1.2-1.3**× under mixed-precision training scenarios. Our system also delivers **2.6-7.6**× speedup to relational graph convolution workloads over DGL, PyG and Graphiler. Code is publicly released at https://github.com/mit-han-lab/torchsparse.

2 BACKGROUND AND MOTIVATION

Without loss of generality, we use point cloud workloads to illustrate the computation pattern of sparse convolution. A point cloud sparse tensor can be defined as an unordered set of points with features: $\{(\boldsymbol{p}_j, \boldsymbol{x}_j)\}$. $\boldsymbol{p}_j$ is the quantized coordinates for the j^{th} point in the D-dimensional space $\mathbb{Z}^D$. $\boldsymbol{x}_j$ is its C-dimensional feature vector in $\mathbb{R}^C$. Coordinate quantization is done through $\boldsymbol{p} = \lfloor \boldsymbol{p}_i^{(\text{raw})} / \boldsymbol{v} \rfloor$, where $\boldsymbol{v}$ is the voxel size vector. `Unique` operation is further applied to all quantized coordinates. For example, in CenterPoint [53], the point clouds on Waymo [38] are quantized using $\boldsymbol{v}$ = [0.1m, 0.1m, 0.15m]. This means that we will only keep one point within each 0.1m×0.1m×0.15m grid.

2.1 Definition of Sparse Convolution

Following the notations in [40], 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



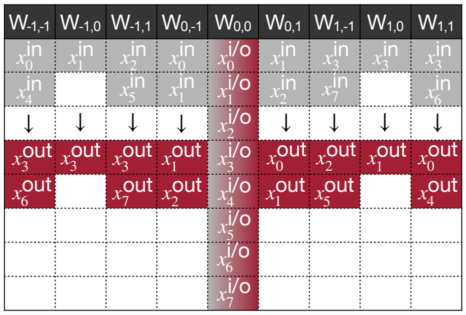
(a) Vanilla weight-stationary

(b) Optimized weight-stationary (TorchSparse)

(c) Fetch-on-Demand

(d) Implicit GEMM

Figure 3: Waterfall diagram for different dataflows for sparse convolution on GPU: weight-stationary dataflows (a, b) are easier to implement and maintain but they do not overlap memory access with computation. Both fetch-on-demand and implicit GEMM dataflows require custom MMA routines but are able to hide the memory access time with pipelining.



$W_{-1,-1}$	$W_{-1,0}$	$W_{-1,1}$	$W_{0,-1}$	$W_{0,0}$	$W_{0,1}$	$W_{1,-1}$	$W_{1,0}$	$W_{1,1}$
x_0^{in}	x_1^{in}	x_2^{in}	x_0^{in}	$x_0^{\text{i/o}}$	x_1^{in}	x_3^{in}	x_3^{in}	x_3^{in}
x_4^{in}		x_5^{in}	x_1^{in}	$x_1^{\text{i/o}}$	x_2^{in}	x_7^{in}		x_6^{in}
↓	↓	↓	↓	$x_2^{\text{i/o}}$	↓	↓	↓	↓
x_3^{out}	x_3^{out}	x_3^{out}	x_1^{out}	$x_3^{\text{i/o}}$	x_0^{out}	x_2^{out}	x_1^{out}	x_0^{out}
x_6^{out}		x_7^{out}	x_2^{out}	$x_4^{\text{i/o}}$	x_1^{out}	x_5^{out}		x_4^{out}
				$x_5^{\text{i/o}}$				
				$x_6^{\text{i/o}}$				
				$x_7^{\text{i/o}}$				

Figure 4: Illustration of the gather-GEMM-scatter dataflow for Figure 2 workload: we first gather input features according to $\mathcal{M}_\delta$ for each weight δ, then perform GEMM or batched GEMM, and finally scatter the results back to output locations given in $\mathcal{M}_\delta$.

(Figure 2) on the k^{th} output point is defined as:

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

where $\boldsymbol{p}_j \in \boldsymbol{P}^{\text{in}}, \boldsymbol{q}_k \in \boldsymbol{P}^{\text{out}}$, $1[\cdot]$ is a binary indicator, s is the stride and $\boldsymbol{W}_\delta \in \mathbb{R}^{C_{\text{in}} \times C_{\text{out}}}$ corresponds to the weight matrix for kernel offset $\delta \in \Delta^D(K)$.

2.2 Sparse Convolution Dataflows on GPUs

Current implementations of sparse convolution on GPUs can be categorized into three distinct dataflows (Figure 3). The first is the *gather-GEMM-scatter* approach, which is weight-stationary and was inspired by early explicit `im2col` attempts [23] for convolution implementation. The second dataflow is the *fetch-on-demand* approach, which is a kernel fusion version of gather-GEMM-scatter. Finally, the *implicit GEMM* approach is an output-stationary alternative inspired by its dense counterpart [11].

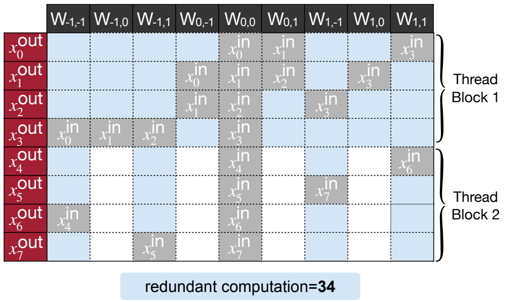
2.2.1 Gather-GEMM-Scatter Dataflow. Early sparse convolution implementations utilized a gather-GEMM-scatter dataflow [18, 50]. This dataflow is weight-stationary and features an outer host loop over K^D kernel offsets. For each offset $\boldsymbol{\delta} \in \Delta^D(K)$, we compute maps $\mathcal{M}_\delta = \{(\boldsymbol{p}_j, \boldsymbol{q}_k) | \boldsymbol{p}_j = s\boldsymbol{q}_k + \boldsymbol{\delta}\}$, as shown in Figure 4. We gather all input features $\boldsymbol{x}_j^{\text{in}}$, resulting in a $|\mathcal{M}_\delta| \times C_{\text{in}}$ matrix in DRAM, and multiply it by weight $\boldsymbol{W}_\delta \in \mathbb{R}^{C_{\text{in}} \times C_{\text{out}}}$. Finally, we scatter the results back to output positions $\boldsymbol{x}_k^{\text{out}}$ according to $\mathcal{M}_\delta$. For example, since $\boldsymbol{p}_0 = 1 \times \boldsymbol{q}_1 + (-1, -1)$ and $\boldsymbol{p}_4 = 1 \times \boldsymbol{q}_5 + (-1, -1)$, $\mathcal{M}_{-1,-1} = \{(\boldsymbol{p}_0, \boldsymbol{q}_1), (\boldsymbol{p}_4, \boldsymbol{q}_5)\}$. We gather $\boldsymbol{x}_0^{\text{in}}$ and $\boldsymbol{x}_4^{\text{in}}$, multiply them by $\boldsymbol{W}_{-1,-1}$, and scatter the results back to $\boldsymbol{x}_1^{\text{out}}$ and $\boldsymbol{x}_5^{\text{out}}$. A variant of this dataflow [40] aims to reduce both computation and data movement time by fusing and reordering memory accesses and grouping computation for different weights.

Gather-GEMM-scatter is straightforward to implement. Following feature gathering, computation for each offset $\boldsymbol{\delta}$ involves a dense matrix multiplication, which can be handled by existing vendor libraries like cuBLAS and cuDNN. Only scatter and gather operations need to be optimized in CUDA. However, this dataflow is fundamentally inefficient due to the lack of overlap between computation and memory access, as illustrated in Figure 3a,b. It is thus impossible to hide data orchestration latency with pipelining.

2.2.2 Fetch-On-Demand Dataflow. The gather-GEMM-scatter implementation requires three separate CUDA kernel calls in each host loop iteration over $\boldsymbol{\delta}$. An alternative fetch-on-demand dataflow [12, 50] (named by [40]) merges the gather, matrix multiplication, and scatter kernel calls into a single CUDA kernel. Instead of materializing the $|\mathcal{M}_\delta| \times C_{\text{in}}$ gather buffer in DRAM, it <u>fetches</u> $\{\boldsymbol{x}_j^{\text{in}} | (\boldsymbol{p}_j, \boldsymbol{q}_k) \in \mathcal{M}_\delta\}$ <u>on demand</u> 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 $\{\boldsymbol{x}_k^{\text{out}} | (\boldsymbol{p}_j, \boldsymbol{q}_k) \in \mathcal{M}_\delta\}$ without first instantiating them in a DRAM scatter buffer. Hong *et al.* [21] further improve the vanilla fetch-on-demand dataflow by introducing block fusion, where the *sequential* host loop over $\boldsymbol{\delta}$ is converted to a *parallel* thread block dimension. As such, the computation of all $\boldsymbol{\delta}$s is merged into a single kernel. Similar to gather-GEMM-scatter (without adaptive grouping in [40]), the fetch-on-demand dataflow has *zero* redundant computation. It further overlaps computation with memory access and saves DRAM writes to gather and scatter buffers.

However, it cannot save any DRAM write to the final output tensor, which means $\frac{\sum_\delta |\mathcal{M}_\delta|}{N_{\text{out}}} \times$ (4×-10× in real workloads, since each point typically has 4-10 neighbors) larger write-back traffic than the theoretical optimal value. Furthermore, the block-fused fetch-on-demand dataflow [21] suffers from write-back contentions between different threads. For example, both $\boldsymbol{W}_{-1,0}$ and $\boldsymbol{W}_{-1,1}$ in Figure 4 may attempt to write back to $\boldsymbol{x}_3^{\text{out}}$. Therefore, it is necessary to introduce atomic operations to serialize all DRAM writes to the same location. Since gather and scatter operations are now combined into GEMM, the entire computation kernel in the fetch-on-demand dataflow must be implemented in CUDA. This is more complex than the gather-GEMM-scatter approach.

2.2.3 Implicit GEMM Dataflow. Very recently, SpConv v2 [49, 50] extends the well-known implicit GEMM formulation [11, 24] for 2D convolution to 3D sparse convolution. As in Figure 5, the sparse

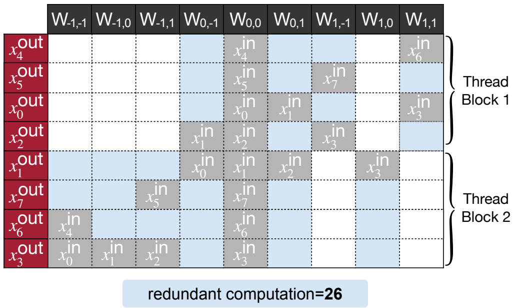


	$W_{-1,-1}$	$W_{-1,0}$	$W_{-1,1}$	$W_{0,-1}$	$W_{0,0}$	$W_{0,1}$	$W_{1,-1}$	$W_{1,0}$	$W_{1,1}$
x_0^{out}					x_0^{in}	x_1^{in}			x_3^{in}
x_1^{out}				x_0^{in}	x_1^{in}	x_2^{in}		x_3^{in}	
x_2^{out}				x_1^{in}	x_2^{in}		x_3^{in}		
x_3^{out}	x_0^{in}	x_1^{in}	x_2^{in}		x_3^{in}				
x_4^{out}					x_4^{in}				x_6^{in}
x_5^{out}					x_5^{in}		x_7^{in}		
x_6^{out}	x_4^{in}				x_6^{in}				
x_7^{out}			x_5^{in}		x_7^{in}				

Figure 5: Illustration of the unsorted implicit GEMM dataflow for Figure 2 workload: each gray 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 4 threads (4 rows).

	$W_{-1,-1}$	$W_{-1,0}$	$W_{-1,1}$	$W_{0,-1}$	$W_{0,0}$	$W_{0,1}$	$W_{1,-1}$	$W_{1,0}$	$W_{1,1}$		
x_0^{out}	0	0	0	0	1	1	0	0	1	25	3rd
x_1^{out}	0	0	0	1	1	1	0	1	0	58	5th
x_2^{out}	0	0	0	1	1	0	1	0	0	52	4th
x_3^{out}	1	1	1	0	1	0	0	0	0	464	8th
x_4^{out}	0	0	0	0	1	0	0	0	1	17	1st
x_5^{out}	0	0	0	0	1	0	1	0	0	20	2nd
x_6^{out}	1	0	0	0	1	0	0	0	0	272	7th
x_7^{out}	0	0	1	0	1	0	0	0	0	80	6th

(a) Bitmask in SpConv v2.



	$W_{-1,-1}$	$W_{-1,0}$	$W_{-1,1}$	$W_{0,-1}$	$W_{0,0}$	$W_{0,1}$	$W_{1,-1}$	$W_{1,0}$	$W_{1,1}$
x_4^{out}					x_4^{in}				x_6^{in}
x_5^{out}					x_5^{in}		x_7^{in}		
x_0^{out}					x_0^{in}	x_1^{in}			x_3^{in}
x_2^{out}				x_1^{in}	x_2^{in}		x_3^{in}		
x_1^{out}				x_0^{in}	x_1^{in}	x_2^{in}		x_3^{in}	
x_7^{out}			x_5^{in}		x_7^{in}				
x_6^{out}	x_4^{in}				x_6^{in}				
x_3^{out}	x_0^{in}	x_1^{in}	x_2^{in}		x_3^{in}				

(b) Sorting reduces redundant computation in SpConv v2.

Figure 6: SpConv v2 sorts the input bitmasks and reorders the computation accordingly. White grids are skipped zero computation. Consequently, redundant computation is reduced from 34 MACs (Figure 5) to 26 for the Figure 2 example.

convolution workload in Figure 2 is equivalent to a dense GEMM $\boldsymbol{X}^{\text{out}} = \boldsymbol{X}^{\text{im2col-in}}_{M\times K} \times \boldsymbol{W}_{K\times N}$. Here, $M = N_{\text{out}}$ (number of output points), $N = C_{\text{out}}$, $K = |\Delta^D(K)|C_{\text{in}}$. We visualize matrix $\boldsymbol{X}^{\text{im2col-in}}$ in Figure 5, where we have $N_{\text{out}} \times |\Delta^D(K)|$ grids and each corresponds to C_{in} channels. Gray grids corresponds to input features. For example, output point $\boldsymbol{q}_1$ has four neighbors: $\boldsymbol{p}_0$ (with $\boldsymbol{w}_{0,-1}$, $\boldsymbol{p}_1$ (with $\boldsymbol{w}_{0,0}$), $\boldsymbol{p}_2$ (with $\boldsymbol{w}_{0,1}$) and $\boldsymbol{p}_3$ (with $\boldsymbol{w}_{1,0}$) so the second row in Figure 5 has four gray entries. One can verify that $\boldsymbol{x}_1^{\text{out}} = \boldsymbol{X}_1^{\text{im2col-in}} \times \boldsymbol{W}$ against Figure 2.

Similar to fetch-on-demand, implicit GEMM overlaps computation with memory access (Figure 3). This allows us to hide the memory latency through pipelining. Like `im2col` in 2D convolution, an implicit GEMM implementation is output-stationary. So it achieves the theoretical minimum DRAM write-back traffic. However, despite having lower DRAM traffic compared to fetch-on-demand, implicit GEMM has non-negligible redundant computation. As shown in Figure 5, we assume that each warp contains four threads. All GPU threads within a warp execute in lockstep. Whenever a thread has a non-empty neighbor at weight $\boldsymbol{\delta}$, all threads in the warp will either perform computation or waste cycles for that weight. This leads to 34 redundant computation in Figure 5, which is even more than 22 effective MACs in this example.

To address this issue, SpConv v2 excludes unsorted implicit GEMM in their design space and utilizes bitmask sorting to minimize computation overhead. Following the approach taken by DSTC [45], each output point is assigned a K^D-dimensional bitmask that indicates the presence of its neighbors. These bitmasks are treated as numbers and sorted, and the order of computation for different outputs is adjusted accordingly. For instance, warp 0 calculates $\boldsymbol{x}^{\text{out}}_{0-4}$ in Figure 5, but it calculates $\boldsymbol{x}^{\text{out}}_{4,5,0,2}$ in Figure 6b instead. Thanks to sorting, computation overhead is reduced from 34 MACs to 26 MACs. In practical applications, sorting can reduce redundant computation by up to 3×, but it remains unclear whether this reduction translates into proportional speedups.

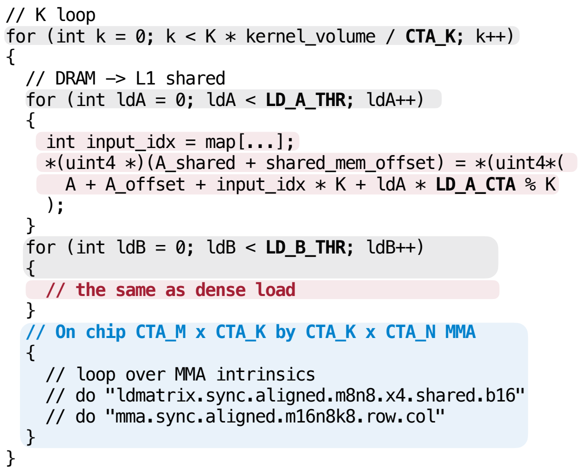
2.3 Motivation

As mentioned above, gather-GEMM-scatter is easy to implement but has poor performance. The more performant dataflows with overlapped computation and memory access cannot be implemented with the help of vendor libraries. Implementing the state-of-the-art implicit GEMM dataflow alone is a daunting task, as demonstrated by the SpConv v2 authors who had to painstakingly re-implement the entire CUTLASS framework from scratch with a custom Python-based template metaprogrammer [48]. The resulting code base has over **40,000** lines of code which increases the risk of errors for developers. This also makes it challenging for the community to explore a wider design space for sparse point cloud convolution kernels, hindering further performance improvements.

Therefore, in TorchSparse++, we want to first demonstrate in Section 3 that highly efficient dataflows with overlapped computation and memory access can be generated with a relatively low engineering complexity (comparable to implementing gather-GEMM-scatter). With the efficient kernel generator as a cornerstone, we further showcase in Section 4 that the design space for sparse point cloud convolution could be significantly extended, and there exists solutions that are up to **1.7**× faster in inference, **1.3**× faster in training compared with the incumbent state-of-the-art within this vast space. Tackling a fundamentally *sparse* workload, we also challenge traditional thinking on *dense* GPU kernel design. Our research reveals that typical first-order performance indicators, such as total computation, DRAM access, or even total runtime for all sparse convolution **computation kernels**, cannot accurately reflect the **end-to-end runtime** of sparse point cloud workloads. This is because sparse workloads require expensive mapping operations. On top of this observation, we will further demonstrate that end-to-end

Table 1: Different sparse convolution dataflows in Section 2 can be mapped onto GPUs as dense GEMM with sparse global memory iterators.

	DRAM → L1 SRAM	MMA	RF → DRAM
GEMM	dense	dense	dense
gather-GEMM-scatter	dense + gather	dense	dense + scatter
fetch-on-demand	**sparse**	dense	**sparse**
implicit GEMM	**sparse**	dense	dense



```
// K loop
for (int k = 0; k < K * kernel_volume / CTA_K; k++)
{
  // DRAM -> L1 shared
  for (int ldA = 0; ldA < LD_A_THR; ldA++)
  {
    int input_idx = map[...];
    *(uint4 *)(A_shared + shared_mem_offset) = *(uint4*(
      A + A_offset + input_idx * K + ldA * LD_A_CTA % K
    );
  }
  for (int ldB = 0; ldB < LD_B_THR; ldB++)
  {
    // the same as dense load
  }
  // On chip CTA_M x CTA_K by CTA_K x CTA_N MMA
  {
    // loop over MMA intrinsics
    // do "ldmatrix.sync.aligned.m8n8.x4.shared.b16"
    // do "mma.sync.aligned.m16n8k8.row.col"
  }
}
```

Figure 7: We introduce Sparse Kernel Generator, a code generator that integrates on-chip MMA subroutines from [4] directly at the source code level, unlocking the potential of using *dense, fixed shape* tensor compiler to generate programs for *sparse, dynamic shape* workloads. Gray: constant code, red: fixed metaprogramming template, blue: generated automatically by existing tensor compiler for each tile size.

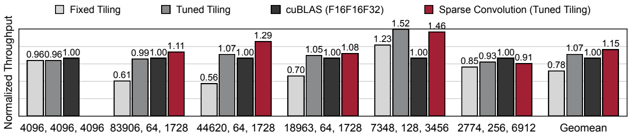


Figure 8: For sparse convolution workloads (MinkUNet on SemanticKITTI), it is possible for our template to achieve or even exceed cuBLAS utilization for the equivalent-sized GEMM problem by tuning *only tiling size parameters.*

optimal dataflows could sometimes choose configurations with up to **6×** computation overhead and **4×** larger DRAM footprint.

3 SPARSE KERNEL GENERATOR

In this section, we introduce the Sparse Kernel Generator, which is a metaprogrammer that can efficiently generate sparse convolution GPU kernels. Existing metaprogrammers, such as TVM [4], are designed to generate optimized GPU computing schedules for *dense* and *fixed-shape* workloads. However, point cloud workloads are naturally *sparse* and have *dynamic shapes*.

3.1 Dense to Sparse Adaptation

Leveraging the information from Section 2, we establish the relationship between sparse convolution and dense GEMM kernels, as summarized in Table 1. We show that the fetch-on-demand and implicit GEMM dataflows with their overlapped memory access and computation can be seen as generalized GEMM kernels with sparse DRAM loading and write-back iterators. Take implicit GEMM as an example, we start from its equivalent-sized dense GEMM workload in Section 2.2.3. We notice that position (i, j) in $\boldsymbol{X}^{\text{im2col-in}}$ is mapped to position $(\mathcal{M}_{i,j/C_{\text{in}}}, j\%C_{\text{in}})$ in $\boldsymbol{X}^{\text{in}}$. Here $\mathcal{M}_{N_{\text{out}}\times|\Delta^D(K)|}$ is the output-stationary representation of the maps defined in Section 2.2.1. For the n^{th} output point, if its k^{th} neighbor is non-empty, then $\mathcal{M}_{n,k}$ is the index of this neighbor; otherwise $\mathcal{M}_{n,k} = -1$. For example, in Figure 5, $\mathcal{M}_{2,3} = 1$ since the fourth neighbor of $\boldsymbol{x}_2^{\text{out}}$ is $\boldsymbol{x}_1^{\text{in}}$. Here we assume indices start from 0. By introducing this one level of indirect addressing, we can easily transition from a dense GEMM to a sparse implicit GEMM when loading data from DRAM to L1 shared SRAM. Since the DRAM→L1 memory access to $\boldsymbol{W}$ is dense, one can reuse the CUDA code segment for 2^{nd} operand loading in dense GEMM. Based on this formulation, as in Figure 7, a sparse convolution kernel can then be decomposed into three parts. The **gray** code is *always constant*. **Blue** code depends on the tile sizes and can be automatically generated by the existing compilers [4]. The **red** code cannot be generated by existing dense tensor compilers due to sparsity, but it can be generated from a *fixed* template that only takes in tiling sizes as input parameters. Consequently, we only need to manually implement the short **red** code template and a TensorIR [13] template that outputs the **blue** on-chip MMA subroutine, which only takes hundreds of lines of code (orders of magnitude cheaper than the SpConv v2 code generator).

For simplicity, we did not visualize performance optimization techniques such as double buffering and pipeling in Figure 7. However, these techniques will not impact the design of our code generator. Similar analysis and code transformation can also be applied to the fetch-on-demand dataflow.

3.2 Static to Dynamic Adaptation

Thanks to the adaptation described in Section 3.1, we can now easily implement sparse convolutions in dataflows with overlapped computation and memory access. However, the simplicity of the code generator comes at the cost of a reduced design space. Our Sparse Kernel Generator only allows the *tiling sizes* to be tuned, while leaving most of the dimensions in the tensor program design space to be fixed (*e.g.* the order and split of the loop nests). Fortunately, we argue that such reduced design space does not compromise the performance. We present an idealized experiment in Figure 8. We manually traverse all possible tile sizes for different layers in MinkUNet [12] on SemanticKITTI [2] and apply compile-time constant folding to maximize performance. We benchmark the resulting sparse kernel with the lowest latency against cuBLAS, which runs an equivalent-sized GEMM problem due to the lack of sparsity support. It turns out that we can achieve > 100% cuBLAS utilization on average by only tuning tile sizes. Notably, for the last workload, the equivalent-sized dense GEMM problem can run at ≈90% device utilization on RTX 3090. If we ignore redundant

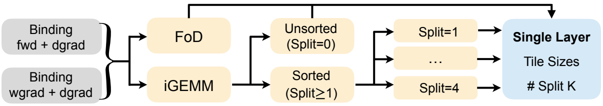


Figure 9: Overview of TorchSparse++ design space.

computation (Figure 5), it is safe to assert that extending the design space beyond tile sizes will not significantly improve final performance on this workload.

Despite achieving encouraging results in the idealized experiment, it remains challenging to transfer the performance to real systems. Unlike dense workloads, each sparse point cloud sample has a different shape in terms of the number of points. Precompiling constant-folded kernels for all possible workloads, as is done by TVM and TensorRT in the dense domain, is impossible for us. Naively unfold the constants in fixed shape kernels and revert them back to workload shape parameters will degrade the performance by up to **1.7**×. This totally undermines the good results achieved in Figure 8. Worse still, the first **red** instruction in Figure 7 now requires explicit boundary check in flexible shape kernels, which brings up to **1.35**× performance overhead as well.

To this end, we present two simple yet effective strategies to address these two performance roadblocks.

We first pinpoint that the slow addressing of $\boldsymbol{X}^{\text{in}}$ is the reason why constant unfolding ruins the performance. Unlike in dense GEMM, accessing $\boldsymbol{X}^{\text{in}}$ requires two inefficient division and modulo operations with C_{in} as an operand, which are necessary just for addressing. This impacts the efficiency since C_{in} is stored in the RF and has an access latency no shorter than L1 on GPUs. Worse still, accesses to $\boldsymbol{X}^{\text{in}}$ are located in the innermost layer of the long K loop ($|\Delta^D(K)| \times C_{\text{in}}$, ranging from 1728 to 6912 in Figure 8). Fortunately, we notice that most of the addressing computation is irrelevant to the innermost loop variable `ldA` in Figure 7. Therefore, it is possible for us to lift the loop invariants out of the loop. For real tiling sizes with `LD_A_THR=4` and 8, this at least reduces addressing cost by 4-8×. We further analyze the template and perform loop invariant hoisting wherever possible. Ablation studies in Section 6.2 shows that addressing simplification can fully close the up to 1.7× constant unfolding overhead.

Likewise, among all boundary checks in the dynamic shape kernel, the one for accessing `map` within the innermost `ldA` loop is the most time-consuming. Although loop invariant hoisting does not apply in this case, we can solve this issue by padding the first dimension of `map` to be a multiple of `cta_M`. With this simple modification, no boundary check on `map` access in Figure 7 is required since we can ensure that every access stays within bounds. With that reduced control flow overhead, we close the final 1.14-1.35× performance gap between fixed and dynamic shape kernels.

4 SPARSE AUTOTUNER

Based on the simple yet powerful Sparse Kernel Generator, we present Sparse Autotuner. It first significantly enlarges the design space of existing libraries (illustrated in Figure 9) and then applies group-based configuration tuning across this enlarged space.

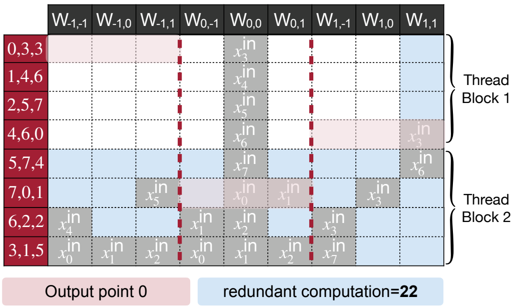
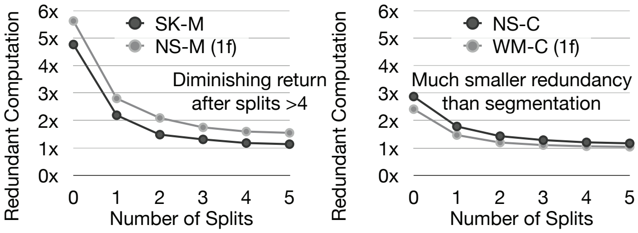


Figure 10: We extend the implicit GEMM design space by introducing arbitrary number of mask splits. Compared with Figure 6b (1 split), splitting the mask into three parts further reduces redundant computation and increases parallelism.



(a) Segmentation Workloads (b) Detection Workloads

Figure 11: A large design space on number of splits in implicit GEMM is beneficial: (a) redundant computation in segmentation workloads continues to drop quickly until splits = 5; (b) redundant computation in detection workloads at split = 0 (unsorted) is acceptable on high-parallelism devices.

4.1 Design Space Augmentation

Thanks to the simplicity of Sparse Kernel Generator, we can easily expand our design space. Since the generator can produce fetch-on-demand kernels, we can effortlessly incorporate this dataflow in our designs. Besides, for implicit GEMM, number of splits (Figure 10) is an important tunable dimension in the implicit GEMM dataflow that was previously overlooked. Similar to the SplitK technique [24] in dense GEMM kernel design, one could split the sequential K loop in Figure 7 into s parts. By doing so, each split (whose K loop is now s× shorter) can compute in parallel and write to a separate DRAM buffer. These partial sums are later reduced by a summation kernel to produce the final result. We also reorder the computation in each split following Figure 6, which involves argsorting s individual bitmasks and reordering the `map` accordingly. For example, after reordering, the first row calculates part of $\boldsymbol{x}_0^{\text{out}}, \boldsymbol{x}_3^{\text{out}}, \boldsymbol{x}_3^{\text{out}}$, while the full feature of $\boldsymbol{x}_0^{\text{out}}$ is calculated in the 1st, 4th, 6th rows by two thread blocks collaboratively. As such, there are more common zero neighbors for each thread block and the redundant computation is further reduced from 26 in Figure 6 to 22 in Figure 10. When integrating support for arbitrary split implicit GEMM, we notice that it is beneficial to reorder the `map` in an offline manner for a similar reason in Section 3.2.

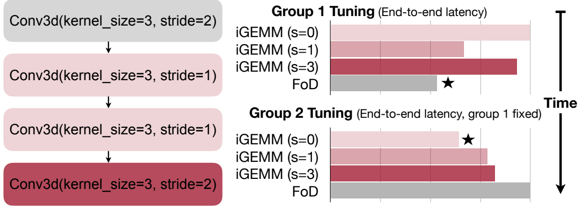
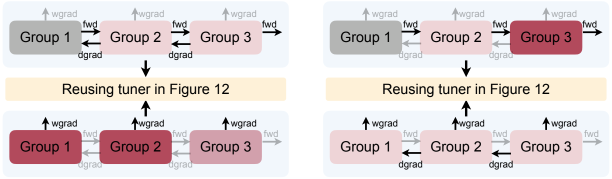


Figure 12: Group-based autotuning: Layers using the same maps will be assigned to the same group. After group partition, we exhaustively traverse all choices in our design space in a group-by-group manner and selects the best group configuration that leads to the lowest end-to-end latency.



(a) Binding fwd-dgrad (for **low-end** devices) (b) Binding dgrad-wgrad (for **high-end** devices)

Figure 13: Parameter binding in training tuner: we propose to partially decouple the dataflow parameters for `forward`, `dgrad` and `wgrad` kernels in training, which leads to up to 10% improvement in end-to-end training time.

Conventionally, dense GPU kernel design is often guided by first-order performance approximation (*e.g.* computation and DRAM footprint). Following these proxies, it seems to be reasonable to eliminate split = 0 (unsorted implicit GEMM in Figure 5) due to its large redundant computation. Split > 2 should also be eliminated since it incurs much larger DRAM write back traffic. In fact, such prematured optimizations lead to the restricted design space in SpConv v2. However, we argue in Figure 11 that it is beneficial to have a larger design space that includes many first-order suboptimal solutions. On the one hand, the redundant computation in both segmentation and detection workloads keeps dropping until $s = 5$. The difference in computation overhead between $s = 2$ and $s = 4$ can still be up to 1.2× for detection and 1.3× for segmentation. Thus, for devices with limited parallelism, it is beneficial to increase the number of splits despite increased DRAM traffic. On the other hand, when running detection workloads on devices with high parallelism, a 2.4-2.9× computation overhead for the unsorted dataflow in Figure 5 is completely acceptable. We will demonstrate in Table 3 and Table 4 that kernels for detection will not run faster despite having $\sim$ 2× lower computation overhead on RTX 3090, which has an ample 71 TFLOPS FP16 peak throughput.

4.2 Group-Based Configuration Tuning

To this end, we designed a *sparse* and *dynamic shape* kernel generator with minimal help from *dense* and *fixed shape* tensor compilers. By doing so, we obtain high-performance sparse convolution kernels with different dataflows (*e.g.* fetch-on-demand and implicit GEMM) and augments the design space of implicit GEMM itself by introducing arbitrary number of mask splits. However, no dataflow is perfect for all workloads. As in Section 2, fetch-on-demand has zero redundant computation but suffers from large DRAM scattering traffic, while implicit GEMM has the exact opposite property. Similarly, there is no single set of parameters that works for each dataflow. For example, the number of splits s in implicit GEMM reflects the tradeoff between redundant computation and control flow overhead (*e.g.* sorting s individual bitmasks and reordering the maps). Therefore, the enlarged design space necessitates the design of an autotuning system that can automatically determine the optimal dataflow and dataflow-specific parameters for different workloads.

To determine the optimal dataflow for different layers, we divide all layers into different groups (illustrated in Figure 12). All layers within each group use the same input-output mappings (*maps*) and are forced to execute the same dataflow. This is because different dataflows require different map structures. Implementations such as gather-GEMM-scatter and fetch-on-demand require the maps to be stored in a weight-stationary order, represented as $\mathcal{M}_{\boldsymbol{\delta}} = \{(\boldsymbol{p}_j, \boldsymbol{q}_k) | \boldsymbol{p}_j = s\boldsymbol{q}_k + \boldsymbol{\delta}, \boldsymbol{q}_k \in \boldsymbol{P}^{\text{out}}\}$, which makes it difficult to infer all the neighbors of an output point (required by implicit GEMM). On the other hand, the implicit GEMM implementation, stores the maps in an output-stationary order, represented as $\mathcal{M}_k = \{\boldsymbol{p}_k^{(\delta)} | \boldsymbol{p}_k^{(\delta)} = s\boldsymbol{q}_k + \boldsymbol{\delta}, \boldsymbol{\delta} \in \Delta^D(K)\}$, which makes it difficult to infer all the inputs that use the same weight (required by the other two dataflows). It would incur significant overhead ($\sim$ latency of up to 3-4 sparse convolution layers within each group!) if we generate maps for all dataflows but only use one of them at runtime. Therefore, allowing intra-group heterogeneous dataflow selection is not desired. After group partition, we apply a group-level exhaustive search on a random subset of the target workload (*e.g.* 100 scenes on the Waymo dataset). Since the execution time of each group is independent of the others, we tune the dataflow parameters in a greedy manner. We iterate over all possible choices for the k^{th} group based on the optimally-tuned configurations for the 1^{st} to $(k-1)^{\text{th}}$ groups, using default parameters for all subsequent groups. This approach effectively reduces the tuner complexity from exponential to linear[*] and allows us to complete tuning within 2 minutes for most workloads. Considering that the tuned schedule could be reused for millions of scenes in real-world ADAS applications during inference, the cost is clearly justifiable.

We further extend Sparse Autotuner to support training workloads. The most straightforward design assumes that the back propagation kernels (*i.e.* `dgrad` for feature map gradient calculation and `wgrad` for weight gradient calculation) share the same dataflow parameters as the `forward` kernel. However, as analyzed in Section 6.1, such design incurs up to 10% performance regression in end-to-end training. Naively decoupling the tuning process for training workloads leads to an unacceptable $O(K^3)$ tuning complexity, with K being the size of our design space. To address this complexity issue, we partially bind dataflow parameters for `forward`, `dgrad`, and `wgrad` kernels. We propose two binding schemes: the **workload-pattern oriented** scheme binds the dataflow parameters for `forward` and `dgrad` kernels while allowing `wgrad` kernels to be tuned separately,

[*] Measuring end-to-end latency during tuning is necessary because in U-net structured models, some groups may contain layers that are not consecutive in the original model.

reducing the tuning complexity to $O(K^2)$ and minimizing the total latency for all sparse convolution kernels. We also propose the **sparse-mapping oriented** scheme, which binds dgrad and wgrad kernels together since they share the same maps, minimizing the overhead for map computation. Similar to our observations in inference kernel autotuning, the high-parallelism devices (*e.g.* A100) is far less sensitive to redundant computation than to mapping overhead, while the low-parallelism devices (*e.g.* 2080 Ti) behaves in the exact opposite way. This explains our design choice to use scheme 1 for low-end devices and scheme 2 for more powerful GPUs. As a final remark, we further notice in Figure 13 that the tuning time could be further reduced from $O(K^2)$ to $O(K)$ if we reuse the group-based tuner in Figure 12 twice and skip different parts of the kernels with dummy initializations during tuning.

5 EVALUATION

5.1 Setup

We implement TorchSparse++ in CUDA and PTX assembly based on TorchSparse [40] and compare it with four state-of-the-art sparse convolution libraries MinkowskiEngine 0.5.4 [12], SpConv 1.2.1 [50], TorchSparse [40] (gather-GEMM-scatter) and SpConv 2.3.5 [50] (sorted implicit GEMM, released in April 2023).All systems are integrated into PyTorch 1.12.0 with CUDA 11.7 and cuDNN 8.4.1 and latency numbers are measured on NVIDIA GPUs.

We follow TorchSparse [40] to evaluate all systems on seven representative real world 3D deep learning workloads: MinkUNet [12] (0.5×/1× width, SK-M) on SemanticKITTI [2], MinkUNet (1 or 3 frames, NS-M) on nuScenes-LiDARSeg [3], CenterPoint [53] (10 frames, NS-C) on nuScenes detection and CenterPoint (1 or 3 frames, WM-C) on Waymo Open Dataset [38]. Among these benchmarks, the SemanticKITTI and Waymo datasets are collected using 64-beam LiDAR sensors while the nuScenes dataset are collected using cheaper 32-beam LiDAR sensors. Multi-frame models increase LiDAR density by superimposing history LiDAR scans to the current frame. For detection workloads (CenterPoint), *we only evaluate the runtime of SparseConv layers.*

5.2 Results

Inference. We compare our results with the baseline designs including MinkowskiEngine, SpConv 1.2.1, TorchSparse and SpConv 2.3.5 in Figure 14. All evaluations are done in unit batch size. TorchSparse++ consistently outperforms **all baseline systems** on GPUs with **all architectures** under **three numerical precisions** by a large margin. On cloud Ampere GPUs (A100 and 3090), it achieves **2.9-3.7**×, **3.2-3.3**×, **2.0-2.2**× and **1.4-1.7**× measured end-to-end speedup over the state-of-the-art MinkowskiEngine, SpConv 1.2.1, TorchSparse and SpConv 2.3.5, respectively. We also compare TorchSparse++ with SpConv 2.3.5 on NVIDIA Jetson Orin, an edge GPU platform widely deployed on real-world autonomous vehicles. Our TorchSparse++ is **1.25**× faster than SpConv 2.3.5 on average, while achieving **1.3-1.4**× consistent speedup across all detection workloads that are most time-critical in real ADAS applications. In addition, TorchSparse++ is competitive on legacy GPU architectures (Turing and Pascal), achieving **at least 1.4**×, **1.8**×, **2.4**×, **2.2**× speedup over SpConv 2.3.5, TorchSparse, SpConv 1.2.1 and MinkowskiEngine. Notably, recent advances in point cloud transformers [32, 39, 43] often claim superior accuracy-latency tradeoffs over sparse convolutional backbones implemented with the SpConv v2 backend. With the much faster TorchSparse++ backend, assuming that the 2D part is deployed with TensorRT, the 3-frame CenterPoint model on Waymo is **1.5**× faster than FlatFormer [32] with higher accuracy on Orin.

Chip	RTX 3090	PointAcc	PointAcc-L
System	TorchSparse++	ASIC	ASIC
Cores	328	64^2=4096	128^2=16384
MACs	328×64=20992	4096	16384
Frequency	1.7 GHz	1 GHz	1 GHz
Peak Performance	35.5 TMACS	4 TMACS	16 TMACS
Proj. Latency	31.6 ms	–	17.8 ms

Table 2: We scale up the systolic array of PointAcc [28] to match the peak performance of RTX 3090 and compare our TorchSparse++ against the ASIC accelerator.

Training. We also compare the training performance of our TorchSparse++ and existing systems on A100 and 2080 GPUs in Figure 15. We run the forward and backward pass of all workloads with batch sizes of 2 in mixed precision training (*i.e.* all gradients are calculated in FP16 precision) except for MinkowskiEngine that does not support FP16. We make sure that all workloads evaluated in Figure 15 can reach the same accuracy using the TorchSparse++ backend compared with TorchSparse (for segmentation workloads) and SpConv 2.3.5 (for detection workloads) with FP32 precision. Given the fact that A100 FP16 tensor core arithmetics has **16**× higher throughput compared with FP32 (non-tensor core) computation (312 TFLOPS *vs.* 19.5 TFLOPS), we do not perform FP32 evaluation. As a result, TorchSparse++ is **4.6-4.8**×, **2.5-2.6**× and **1.2-1.3**× faster than MinkowskiEngine, TorchSparse and SpConv 2.3.5 on both Ampere and Turing GPUs. TorchSparse++ paves the way for rapid model iteration for real-world ADAS applications.

Comparison against Accelerators. We further compare the performance of TorchSparse++ on RTX 3090 against a scaled-up version of PointAcc [28] using the SemanticKITTI-MinkUNet workload. The systolic array in PointAcc is enlarged from 64×64 to 128×128 to roughly match the number of MACs (16384 *vs.* 20992) on RTX 3090. The PointAcc memory bandwidth is scaled up accordingly. Since the accelerator adopts IC-OC parallelism, we assume that the scaled PointAcc-L achieves linear speedup if the executed layer has large enough input and output channels. We also scale the measured TorchSparse++ latency by 1.7 (clock frequency difference) × 1.3 (peak MACs difference) = 2.2× for fair comparisons. As a result, TorchSparse++ achieves **56% of ASIC speed** on a general-purpose hardware platform with similar computation budget. Notice that we also attempt to make a direct comparison with Mesorasi [15], which codesigns the point cloud convolution algorithm with the hardware architecture. However, its delayed aggregation scheme could only work for convolution operators with *shared* weights for all neighbors. The main workload accelerated in this paper, sparse convolution, is more complicated because it has different weights

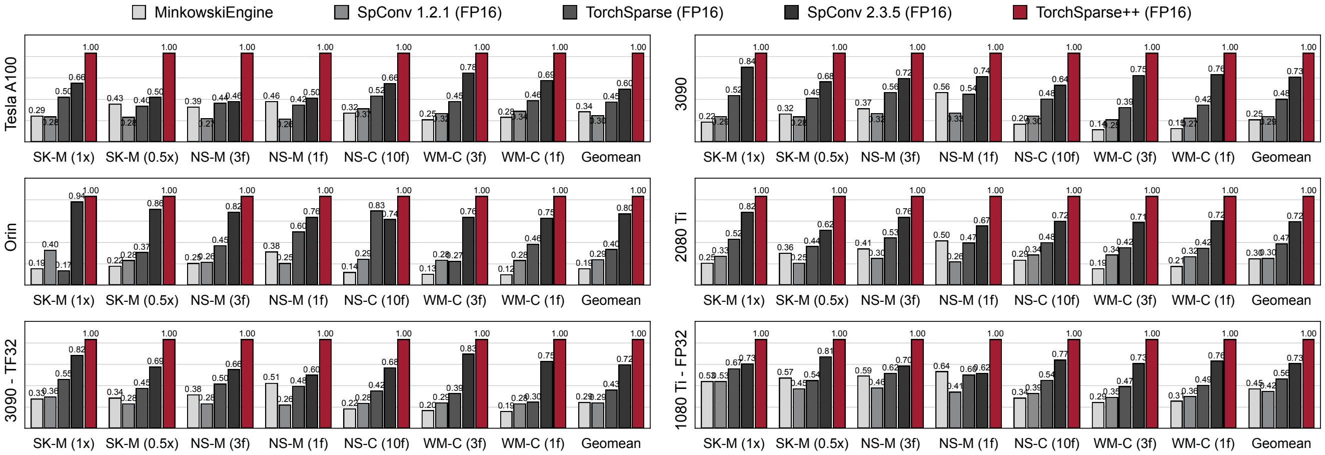


Figure 14: Inference Speedup: TorchSparse++ significantly outperforms existing point cloud inference engines in both 3D object detection and LiDAR segmentation benchmarks across three generations of GPU architecture (Pascal, Turing and Ampere) and three precisions (FP16, TF32, FP32). It is up to 1.7× faster than state-of-the-art SpConv 2.3.5 and is up to 2.2× faster than TorchSparse on cloud GPUs. It also improves the latency of SpConv 2.3.5 by 1.25× on Orin.

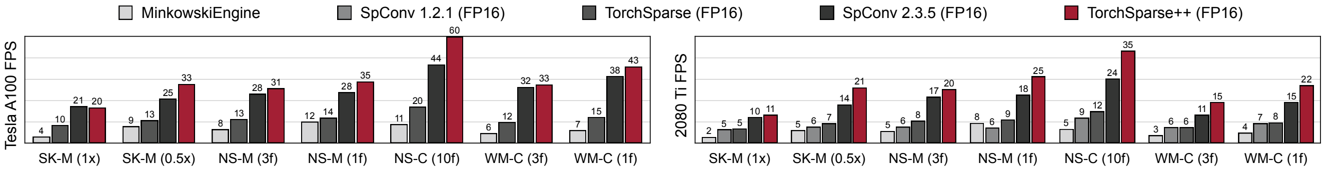


Figure 15: Training Speedup: TorchSparse++ achieves faster FP16 training speed compared with MinkowskiEngine, TorchSparse and SpConv 2.3.5. It is 1.16× faster on Tesla A100, 1.27× faster on RTX 2080 Ti than state-of-the-art SpConv 2.3.5. It also significantly outperforms MinkowskiEngine by 4.6-4.8× across seven benchmarks on A100 and 2080 Ti.

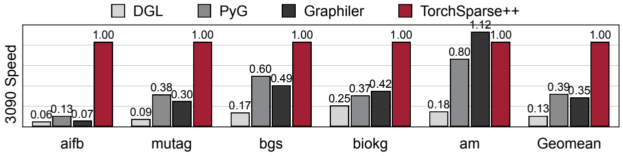


Figure 16: TorchSparse++ is 2.6-7.6× faster than DGL, PyG and Graphiler on five heterogeneous graph benchmarks.

nuScenes-CenterPoint	Unsorted	Split=1	Split=2
RTX 3090 latency (ms)	7.45	8.18	8.81
Orin latency (ms)	21.90	22.20	23.93
Waymo-CenterPoint-1f	Unsorted	Split=1	Split=2
RTX 3090 latency (ms)	8.15	8.91	9.48
Orin latency (ms)	28.68	29.86	33.12

Table 3: End-to-end Latency: Unsorted implicit GEMM is up to 1.2× faster with up to 1.7× redundant computation.

for different neighbors (see Figure 2). Therefore, such comparison might be hard to achieve.

Results on Graph Workloads. . We also implement R-GCN [36] with TorchSparse++ and benchmark it on five representative heterogeneous graph datasets against state-of-the-art graph deep learning systems DGL [44], PyG [16] and the Graphiler [46] compiler. TorchSparse++ is **7.6**×, **2.6**×, **2.9**× faster and **3.4**×, **4.4**×, **5.6**× more memory efficient compared with DGL, PyG and Graphiler.

6 ANALYSIS

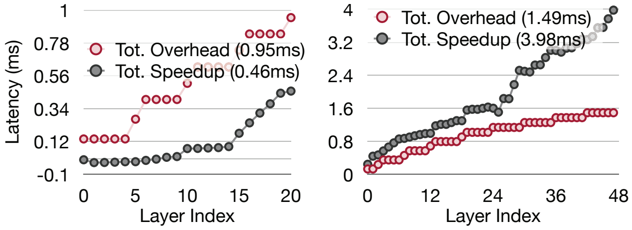
In this section, we present in-depth analysis on the design choices of our Sparse AutoTuner and Sparse Kernel Generator and ablate the source of performance gains in Section 5.

6.1 Design Space of Sparse AutoTuner

As discussed in Section 3, the design space of TorchSparse++ is a superset of SpConv v2. We have added several new features to this space, including support for unsorted implicit GEMM, implicit

nuScenes-CenterPoint	Unsorted	Split=1	Split=2
RTX 3090 latency (ms)	3.70	3.68	3.89
Orin latency (ms)	12.57	10.87	12.39
Waymo-CenterPoint-1f	Unsorted	Split=1	Split=2
RTX 3090 latency (ms)	4.60	4.33	4.49
Orin latency (ms)	19.33	17.06	19.75

Table 4: SparseConv Kernel Latency: Unsorted implicit GEMM kernels could be slower than their mask split counterpart, which is the exact opposite of Table 3 results.



(a) Waymo CenterPoint (1 frame) (b) SemanticKITTI MinkUNet

Figure 17: Sorting is able to reduce the computation time, but its overhead outweighs the benefit on detection workloads.

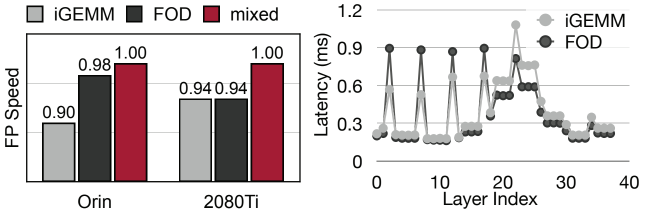
GEMM with an arbitrary number of mask splits (> 2), and the fetch-on-demand dataflow. The flexibility of TorchSparse++ also allows us to explore different dataflow parameter bindings for `forward`, `dgrad`, and `wgrad` computation. As such, we challenge conventional designs that shares the same dataflow parameters across all kernels. In the following two subsections, we will evaluate the effectiveness of all these new design choices in TorchSparse++.

Effectiveness of unsorted implicit GEMM.. We first demonstrate the efficacy of unsorted implicit GEMM dataflow (Figure 5) against the sorted implicit GEMM dataflow in SpConv v2. As in Table 3, the unsorted dataflow is consistently faster on both server and edge GPUs. We further present runtime comparison of all *sparse convolution kernels* between unsorted and sorted dataflows in Table 4. Interestingly, if we only consider the runtime of convolution kernels, the sorted dataflow is indeed faster. However, the latency difference between Table 3 and Table 4 reveals the fact that sparsity-incurred mapping overhead (*e.g.* obtaining the bitmask, sorting the bitmask, performing bitmask reduction and reordering the maps) in the sorted dataflow is non-negligible.

Moreover, Figure 17 shows the layerwise comparison of these two versions of TorchSparse++, in which the gain from reduction in computation is overweighed by the overhead of sorting itself on Waymo object detection. However, sorting does show an advantage on a larger segmentation model (MinkUNet) on the SemanticKITTI benchmark.Our observation challenges the design principle of SpConv v2, which is to use *amount of computation* as a first-order approximation for end-to-end performance. It also nullifies the assumption that *faster computation kernel* is equivalent to *better end-to-end performance.*

Number of splits	{1}	{1, 2}	{0, 1, 2, 3, 4}
FP16 latency (ms)	16.01	15.23	14.48
TF32 latency (ms)	26.65	23.28	21.28
FP32 latency(ms)	35.47	29.92	25.75

Table 5: We evaluated the performance of a SemanticKITTI-MinkUNet workload on an RTX 3090 and found that expanding the design space of implicit GEMM by increasing the number of splits led to up to 1.4× improvement compared to the default setting (split=1) in SpConv v2.



(a) Effectivness of FOD (b) 2080 Ti Layerwise Breakdown

Figure 18: Fetch-on-demand and implicit GEMM dataflows are complementary to each other on FP32 segmentation workloads. A hybrid dataflow is up to 1.06× faster than the best single dataflow.

Effectiveness of larger split mask design space. We have shown the effectiveness of unsorted implicit GEMM. Additionally, we found that it's also beneficial to have a larger number of splits for segmentation workloads, as demonstrated in Table 5. The parallelism of an implicit GEMM kernel will be increased by $s\times$ with s splits. Because segmentation workloads usually have smaller number of input points, they are more prone to suffer from device under-utilization and increased parallelism will be beneficial. Similarly, the overhead for mapping and partial sum reduction kernels is smaller in segmentation workloads. Significantly reduced computation overhead (Figure 11) further supports the preference for a larger number of splits in these scenarios.

Effectiveness of adding fetch-on-demand. We then choose 1-frame MinkUNet on nuScenes running on RTX 2080 Ti and Orin as a benchmark to demonstrate the efficacy of fetch-on-demand dataflow. As in Figure 18, individually-tuned implicit GEMM and fetch-on-demand dataflows both achieve inferior performance compared with the hybrid dataflow TorchSparse++. We further present the layerwise latency breakdown of the best tuned implicit GEMM and fetch-on-demand configurations in Figure 18b, where we amortize the mapping time to all layers within each layer group (defined in Section 4). The end-to-end performance of fetch-on-demand is notably better than implicit GEMM in decoder layers (*i.e.* layer index > 18) but gets outperformed in downsampling layers, where maps $\mathcal{M}$ could not be reused. This is because implicit GEMM has lower mapping cost while fetch-on-demand computation kernels run faster for the given workload.

Effectiveness of tuner design for training. We finally demonstrate that decoupling dataflow parameters for `forward`, `dgrad` and `wgrad`

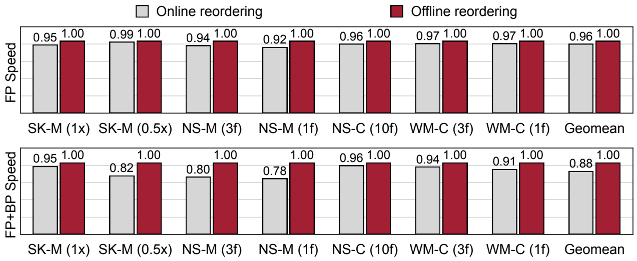


Figure 19: Offline reordering led to a 4% improvement in inference performance and a 12% improvement in training performance compared with online reordering.

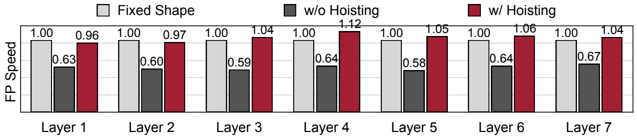


Figure 20: Naively converting fixed shape dense tensor programs to flexible shape sparse convolution kernels will incur 1.5-1.7× runtime overhead due to repetitive pointer calculation. We bridge such huge performance gap via loop invariance hoisting and show that constant folding is unnecessary for high-performance sparse kernels.

kernels could improve the training performance by up to 10% in Figure 22. On both A100 and 2080 Ti, binding parameters for two of the kernels is better than using the same parameters for all three kernels. On A100, binding `dgrad` and `wgrad` is better. This is because such strategy could minimize mapping overhead and there is a drastic performance difference (16×) between tensor cores (which runs computation) and CUDA cores (which runs mapping) on A100. On 2080 Ti, binding `forward` and `dgrad` is better, since the two kernels share the same workload pattern. Given much smaller performance gap between tensor and CUDA cores on 2080 Ti (3×), the additional mapping overhead for decoupled `wgrad` and `dgrad` is acceptable.

6.2 Sparse Kernel Generator

In this section, we present an analysis of the effectiveness of the design choices outlined in Section 3. Our experiments were conducted on 3090 GPUs using FP32 precision for offline reordering and FP16 precision for all other experiments. Our results demonstrate that simplifying control flows and addressing is critical for achieving optimal performance in sparse kernels. Additionally, we found that the conventional wisdom of fusing GPU kernels as much as possible may not always be applicable in the context of sparse computing.

Effectiveness of offline reordering. We present the effectiveness of offline reordering in Figure 19. As described in Section 4, our approach involves reordering computations based on the values of bitmasks in the implicit GEMM dataflow with mask splitting. While conventional wisdom in GPU kernel design suggests *fusing kernels as much as possible* (including reordering in the sparse convolution kernel), our experiments demonstrate that this can lead to a 4-12% *reduction* in end-to-end performance compared to offline reordering. Specifically, when considering the `wgrad` kernels, it is necessary to iterate over the N_{out} dimension in the large and innermost K loop. Online reordering introduces an additional level of indirect addressing to the memory access in the innermost loop. This will disrupt the continuous access pattern and results in a significant slowdown for `wgrad`.

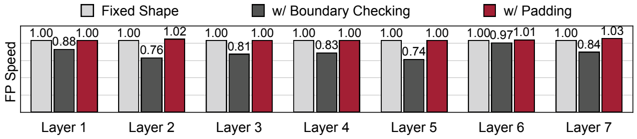


Figure 21: Boundary checking when loading `input_idx` in Figure 7 is a major performance bottleneck that leads to 1.3× latency overhead and can be resolved by offline padding.

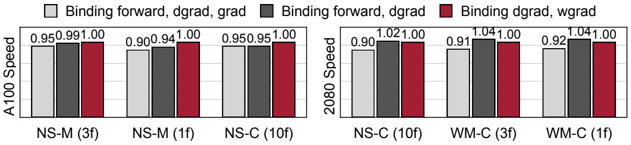


Figure 22: Different from dense kernels, sparse `forward`, `dgrad` and `wgrad` kernels have different preferences for dataflow parameters. Binding hyperparameters for all kernels could hurt the training performance by up to 10%.

Effectiveness of control flow simplification. We use MinkUNet on SemanticKITTI as an example to illustrate the importance of simplifying addressing and control flows. In Figure 20, we evaluate the benefits of loop invariance hoisting. The results show that a naively converted template can be very inefficient. It is up to 1.7× slower than the original fixed shape CUDA kernel. However, with the help of loop invariance hoisting in which we move all the common pointer offsets to the outmost possible loop, we can almost totally eliminate the pointer arithmetic overheads. After applying this technique, our templated CUDA kernel can even run slightly faster than the original fixed shape kernels in 5 of 7 sample workloads. Figure 21 shows the benefits of reducing control flow instructions by padding the `map` in Figure 7. The instructions performing boundary checking can make the kernel up to 1.3× slower. Whereas, after eliminating these control flow instructions, this problem can be well solved with the help of padding.

Effectiveness of adaptive tiling. We experiment with two sets of tiling sizes in TorchSparse++ dependent upon the MACs of the workload. Adaptive tiling provides up to 1.6× speedup to TorchSparse++, compared with fixed tiling version (either always using the small tile sizes or always using the larger tile sizes).

6.3 Discussions

Summary on performance gain. In Figure 23, we present a summary of the performance improvement achieved through the use of our Sparse Kernel Generator and the enlarged design space. Our

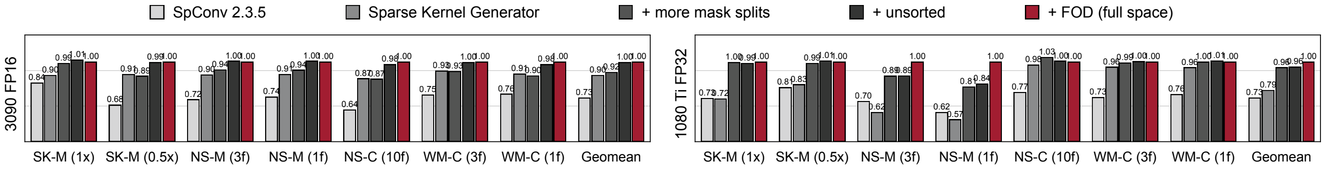


Figure 23: Summary of performance gain from different techniques and the enlarged design space in TorchSparse++.

generator produces high-performance sparse convolution kernels that are 1.1 − 1.2× faster than SpConv 2.3.5, even when using the same dataflow parameters. Remarkably, our code generator comprises only 5% of the lines of code of SpConv 2.3.5's metaprogrammer, which significantly reduces system complexity and enhances programmer productivity. For the enlarged design space, more mask splits are very helpful for *segmentation workloads* and *FP32 precision*, while unsorted implicit GEMM is helpful for *detection workloads* and *FP16 precision*. The efficacy of fetch-on-demand is mainly demonstrated in smaller segmentation workloads (*e.g.* NS-M). These results reinforce that there is no one-size-fits-all strategy for sparse kernel design, and that relying on first-order approximations for end-to-end performance is unreliable.

Insights for microarchitectural improvements. Our TorchSparse++ also provides new insights for future microachitecture design. Our findings indicate that when memory bandwidth is halved on RTX 3090, the latency of the system increases by **1.2**×. In contrast, reducing peak computation throughput by a factor of 2 results in a more substantial slowdown of **1.4**×. Therefore, scaling computation units instead of off-chip memory bandwidth can provide more effective improvements. Moreover, it is apparent from Table 3 and Table 4 that mapping operations account for up to 50% of the total runtime. Leveraging the efficient ASIC design [28] for these operators could significantly enhance the performance of GPUs when executing sparse computation workloads.

Future applications. TorchSparse++ platform presents novel opportunities for enhancing machine learning workloads beyond point clouds and graphs. For instance, in image segmentation [26] and video recognition [33], not all pixels hold equal significance. Hence, the selective computation on a sparse subset of pixels using TorchSparse++ can potentially significantly enhance efficiency. Furthermore, masked autoencoders (MAEs) [20] exhibit inherent sparsity in input patterns during training. While existing approaches already attempt to exploit this sparsity using sparse convolution [22, 42], we posit that TorchSparse++ has the potential to unlock even greater speedups for such workloads.

7 RELATED WORK

Compiler-Based Tensor Program Optimization. Our system benefits from recent advances in tensor program compilation. The pioneering research TVM [4] provides graph-level and operator-level abstractions for deep learning workloads based on the essence of Halide [35]. Based on TVM, AutoTVM [5] automatically discovers the optimal mapping of a fixed-shape tensor program onto the target hardware. Nimble [37] and DietCode [57] are compilers stemmed from 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 [13] proposed a new IR for tensor programs and allows easier tensorization of accelerator primitives. SparseTIR [52] further extended TensorIR to support sparse workloads. Bolt [47] combines the advantages of fully-automatically generated kernels [4] with hand-written subroutines [24] through graph matching.

Point Cloud Accelerators. Deep learning on point clouds has also generated considerable interest in domain-specific accelerator design. Zhu *et al.* [59] proposed a sparsewise dataflow that skips cycles for zero-weight computations and saves energy through gating. Mesorasi [15] co-designed its architecture with the delayed aggregation algorithm to reduce redundant computation in point cloud NNs. More recently, Point-X [55] exploited spatial locality in point clouds through clustering, mapping point clouds into distributed computation tiles. It maximized parallelism and minimized data movement. Additionally, PointAcc [28] mapped all mapping operators in point cloud NNs to a versatile bitonic sorter, making it the first specialized accelerator to support 3D sparse convolution computation. Crescent [14] tamed irregularities in point clouds through approximate neighbor search and selectively elided bank conflicts, while Ying et al. [54] pushed point cloud compression to edge devices through intra- and inter-frame compression.

8 CONCLUSION

We introduce TorchSparse++, a high-performance GPU sparse computation library designed for point cloud and graph deep learning. TorchSparse++ features a highly optimized Sparse Kernel Generator with less than one-tenth of the engineering cost compared with the state-of-the-art system. It further enables us to build an input-aware Sparse Autotuner that selects the best configuration for each layer. TorchSparse++ achieves **1.7-3.3**× inference speedup and **1.2-3.7**× faster training compared to state-of-the-art MinkowskiEngine, SpConv v1/v2, and TorchSparse on seven real-world perception workloads. TorchSparse++ also achieves **2.6-7.6**× speedup over DGL, PyG and Graphiler when running R-GCNs. We hope that TorchSparse++ will facilitate future system and microarchitectural research in sparse computation on 3D data and graphs.

ACKNOWLEDGMENTS

We would like to thank Yan Yan and Bohan Hou for helpful discussions. This work was supported by MIT-IBM Watson AI Lab, MIT AI Hardware Program, MIT-Amazon Science Hub, NSF and Hyundai Motor. Zhijian Liu was partially supported by the Qualcomm Innovation Fellowship.

A ARTIFACT APPENDIX

A.1 Abstract

This artifact contains all the necessary scripts and dependencies to faithfully reproduce the crucial experiments presented in the paper. To successfully run the experiments, a host system with x86_64 CPUs is required, along with at least one NVIDIA GPU with Pascal or newer architecture. Our artifact is provided as two docker files. One for SpConv v1 [50] and TorchSparse [40] and the other for MinkowskiEngine [12], SpConv v2 [49] and TorchSparse++. We made necessary modifications to baseline systems such as MinkowskiEngine and SpConv v1 to make them runnable in the software environment specified in our Dockerfile.

A.2 Artifact check-list (meta-information)

- **Program:** Benchmarking code for all systems and modified source code for baseline systems including MinkowskiEngine [12], SpConv v1 [50], TorchSparse [40];
- **Compilation:** Done in the docker;
- **Model:** MinkUNet [12] and CenterPoint [53];
- **Data set:** SemanticKITTI [2], nuScenes [3], Waymo Open Dataset [38];
- **Run-time environment:** NVIDIA Container Toolkit (nvidia-docker);
- **Hardware:** A host with x86_64 CPUs and at least one NVIDIA GPU in Pascal, Volta, Turing, Ampere or Hopper architectures;
- **Execution:** All benchmarks are executed on NVIDIA GPUs, while some data preprocessing code is executed on the host CPU;
- **Metrics:** Execution time;
- **Output:** Speed of all systems in frames per second (FPS);
- **Experiments:** Inference time measurement for all baseline systems and TorchSparse++ on seven benchmarks (MinkUNet-1× on SemanticKITTI, MinkUNet-0.5× on SemanticKITTI, MinkUNet-1 frame on nuScenes, MinkUNet-3 frame on nuScenes, CenterPoint-10 frame on nuScenes, CenterPoint-1 frame on Waymo, CenterPoint-3 frame on Waymo);
- **How much disk space required (approximately)?:** 100G;
- **How much time is needed to prepare workflow (approximately)?:** Around 2 hours to download the 35G dataset depending on the Internet speed and around 1 hour to build dockers depending on the CPU performance;
- **How much time is needed to complete experiments (approximately)?:** Less than one hour using the ‘fast’ mode, or five hours with complete evaluation;
- **Publicly available?:** Code will be publicly available at the end of the AE process, data cannot be released due to the requirements of dataset providers;
- **Code licenses (if publicly available)?:** MIT;
- **Data licenses (if publicly available)?:** N/A;
- **Workflow framework used?:** Docker;
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.8311888.

A.3 Description

A.3.1 How to access. Our artifact is composed of two parts: the source code and the data. The source code can be accessed through the following doi: https://doi.org/10.5281/zenodo.8311888. After downloading the zip file in this link, please decompress it. Detailed instructions are available in `README.md`. To access the data, please contact Haotian Tang and Shang Yang for a preprocessed subset of all datasets. The users can alternatively download full datasets from SemanticKITTI, nuScenes, Waymo Open Dataset, which takes at least 2TB of disk storage.

A.3.2 Hardware dependencies. Our main experiments are conducted on NVIDIA A100, 3090, Orin (Ampere), 2080 Ti (Turing) and 1080 Ti (Pascal) GPUs. We also tested the artifact on H100 (Hopper) and 4090 (Ada) and it worked well on these newer architectures. If users are unable to access a GPU machine, they can reach out to us via HotCRP, and we will promptly provide them with the necessary access.

A.3.3 Software dependencies. We resolve all software dependencies using the docker environment. The users are expected to have access to the NVIDIA Container Tookit on their machines.

A.3.4 Data sets. We conduct benchmarking experiments on SemanticKITTI [2], nuScenes [3] and Waymo Open Dataset [38]. All these datasets are large-scale autonomous driving benchmarks. Please notice that all datasets prohibit distribution of raw data, so the users are not supposed to publicly distribute our preprocessed data. Additionally, please register at the dataset websites (SemanticKITTI, nuScenes, Waymo Open Dataset) before downloading our preprocessed data.

A.3.5 Models. We provide implementations of MinkUNet [12] and CenterPoint [53] in the artifact.

A.4 Installation

We recommend the users to install the environment and run all experiments in a docker with GPU support. The users can alternatively follow the instructions in our Dockerfile to directly install the dependencies in the host system.

We use the installation process for part 1 as an example. After downloading the repository, please run:

```
cd artifact-p1/docker
docker build -t torchsparsepp-p1 .
docker run -it --runtime=nvidia \
  --gpus all  \
  -v <path-to-benchmark-data>:/workspace/benchmarks \
  -v <path-to-artifact-p1>/evaluation:/workspace/
     evaluation \
  torchsparsepp-p1:latest /bin/bash
```

One only needs to replace `p1` with `p2` in the code snippet to set up part 2.

A.5 Experiment workflow

The users should first symbolic-link `/workspace/benchmarks` (which contains all datasets) to the `/workspace/evaluation` directory in the docker. We then expect the users to run all the experiments with a single line script:

```
OMP_NUM_THREADS=1 python evaluate.py --restart
```

The flag `restart` means that even if there are saved experiment results, the evaluation job will still start from scratch. The users can also add a flag `fast` to perform evaluation on a subset of all benchmarks. This will significantly speedup the experiments. Benchmarking results will be directly printed in the command line.

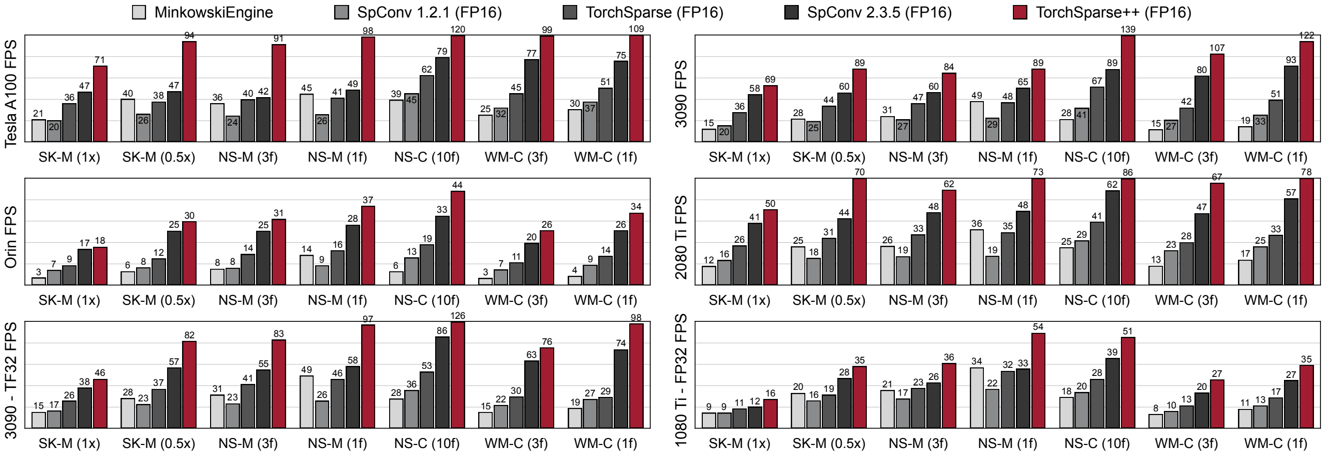


Figure 24: Expected inference throughput of TorchSparse++ and baseline systems at batch size = 1.

A.6 Evaluation and expected results

We provide the expected inference speed on different hardware platforms in Figure 24. Please kindly observe that while the GPU platform may be identical, the absolute frames per second (FPS) numbers can still vary, as the clock frequencies of different GPUs might differ. However, we believe that the relative acceleration ratio should be similar.

A.7 Experiment customization

The users are encouraged to carry out experiments under different data precisions by setting the `precision` flag. Please notice that FP16 is supported for Turing or newer GPUs and TF32 is supported for Ampere or newer GPUs. FP32 is supported for all architectures. The `batch_size` flag controls the inference batch sizes. Model architectures are defined in `core/models` and can also be modified.

A.8 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

REFERENCES

[1] Xuyang Bai, Zeyu Hu, Xinge Zhu, Qingqiu Huang, Yilun Chen, Hongbo Fu, and Chiew-Lan Tai. 2022. TransFusion: Robust LiDAR-Camera Fusion for 3D Object Detection with Transformers. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[2] Jens Behley, Martin Garbade, Andres Milioto, Jan Quenzel, Sven Behnke, Cyrill Stachniss, and Juergen Gall. 2019. SemanticKITTI: A Dataset for Semantic Scene Understanding of LiDAR Sequences. In *IEEE/CVF International Conference on Computer Vision (ICCV)*.

[3] Holger Caesar, Varun Bankiti, Alex H. Lang, Sourabh Vora, Venice Erin Liong, Qiang Xu, Anush Krishnan, Yu Pan, Giancarlo Baldan, and Oscar Beijbom. 2020. nuScenes: A Multimodal Dataset for Autonomous Driving. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[5] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to Optimize Tensor Programs. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[6] Xuesong Chen, Shaoshuai Shi, Benjin Zhu, Ka Chun Cheung, Hang Xu, and Hongsheng Li. 2022. MPPNet: Multi-Frame Feature Intertwining with Proxy Points for 3D Temporal Object Detection. In *European Conference on Computer Vision (ECCV)*.

[7] Yukang Chen, Yanwei Li, Xiangyu Zhang, Jian Sun, and Jiaya Jia. 2022. Focal Sparse Convolutional Networks for 3D Object Detection. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[8] Yukang Chen, Jianhui Liu, Xiaojuan Qi, Xiangyu Zhang, Jian Sun, and Jiaya Jia. 2023. Scaling up Kernels in 3D CNNs. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[9] Ran Cheng, Christopher Agia, Yuan Ren, Xinhai Li, and Bingbing Liu. 2020. S3CNet: A Sparse Semantic Scene Completion Network for LiDAR Point Clouds. In *Conference on Robot Learning(CoRL)*.

[10] Ran Cheng, Ryan Razani, Ehsan Taghavi, Enxu Li, and Bingbing Liu. 2021. (AF)^2-S3Net: Attentive Feature Fusion with Adaptive Feature Selection for Sparse Semantic Segmentation Network. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[11] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. In *Advances in Neural Information Systems (NeurIPS) Workshops*.

[12] Christopher Choy, JunYoung Gwak, and Silvio Savarese. 2019. 4D Spatio-Temporal ConvNets: Minkowski Convolutional Neural Networks. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[13] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, and Tianqi Chen. 2023. TensorIR: An Abstraction for Automatic Tensorized Program Optimization. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[14] Yu Feng, Gunnar Hammonds, Yiming Gan, and Yuhao Zhu. 2022. Crescent: Taming Memory Irregularities for Accelerating Deep Point Cloud Analytics. In *International Symposium on Computer Architecture (ISCA)*. 962–977.

[15] Yu Feng, Boyuan Tian, Tiancheng Xu, Paul Whatmough, and Yuhao Zhu. 2020. Mesorasi: Architecture Support for Point Cloud Analytics via Delayed-Aggregation. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1037–1050.

[16] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

[17] Runzhou Ge, Zhuangzhuang Ding, Yihan Hu, Wenxin Shao, Li Huang, Kun Li, and Qiang Liu. 2021. 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 (CVPR) Workshops*.

[18] Benjamin Graham, Martin Engelcke, and Laurens van der Maaten. 2018. 3D Semantic Segmentation With Submanifold Sparse Convolutional Networks. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[19] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Advances in Neural Information Processing Systems*

(NeurIPS).

[20] Kaiming He, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollár, and Ross Girshick. 2022. Masked Autoencoders Are Scalable Vision Learners. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[21] Ke Hong, Zhongming Yu, Guohao Dai, Xinhao Yang, Yaoxiu Lian, Zehao Liu, Ningyi Xu, and Yu Wang. 2023. Exploiting Hardware Utilization and Adaptive Dataflow for Efficient Sparse Convolution in 3D Point Clouds. In *Conference on Machine Learning and Systems (MLSys)*.

[22] Lang Huang, Shan You, Mingkai Zheng, Fei Wang, Chen Qian, and Toshihiko Yamasaki. 2022. Green Hierarchical Vision Transformer for Masked Image Modeling. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[23] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In *ACM Multimedia*.

[24] Andrew Kerr, Haicheng Wu, Manish Gupta, Dustyn Blasig, Pradeep Ramini, et al. 2022. CUTLASS: CUDA Template Library for Linear Algebra Subroutines. https://github.com/NVIDIA/CUTLASS.

[25] Alex H. Lang, Sourabh Vora, Holger Caesar, Lubing Zhou, and Jiong Yang. 2019. PointPillars: Fast Encoders for Object Detection from Point Clouds. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[26] Xiaoxiao Li, Ziwei Liu, Ping Luo, Chen Change Loy, and Xiaoou Tang. 2017. Not All Pixels Are Equal: Difficulty-Aware Semantic Segmentation via Deep Layer Cascade. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[27] Yanwei Li, Yilun Chen, Xiaojuan Qi, Zeming Li, Jian Sun, and Jiaya Jia. 2022. Unifying Voxel-based Representation with Transformer for 3D Object Detection. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[28] Yujun Lin, Zhekai Zhang, Haotian Tang, Hanrui Wang, and Song Han. 2021. PointAcc: Efficient Point Cloud Accelerator. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[29] Zhijian Liu, Alexander Amini, Sibo Zhu, Sertac Karaman, Song Han, and Daniela Rus. 2021. Efficient and Robust LiDAR-Based End-to-End Navigation. In *IEEE International Conference on Robotics and Automation (ICRA)*.

[30] Zhijian Liu, Haotian Tang, Alexander Amini, Xinyu Yang, Huizi Mao, Daniela Rus, and Song Han. 2023. BEVFusion: Multi-Task Multi-Sensor Fusion with Unified Bird's-Eye View Representation. In *IEEE International Conference on Robotics and Automation (ICRA)*.

[31] Zhijian Liu, Haotian Tang, Shengyu Zhao, Kevin Shao, and Song Han. 2021. PVNAS: 3D Neural Architecture Search with Point-Voxel Convolution. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)* (2021).

[32] Zhijian Liu, Xinyu Yang, Haotian Tang, Shang Yang, and Song Han. 2023. FlatFormer: Flattened Window Attention for Efficient Point Cloud Transformer. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[33] Bowen Pan, Wuwei Lin, Xiaolin Fang, Chaoqin Huang, Bolei Zhou, and Cewu Lu. 2018. Recurrent Residual Module for Fast Inference in Videos. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[34] Charles R Qi, Yin Zhou, Mahyar Najibi, Pei Sun, Khoa Vo, Boyang Deng, and Dragomir Anguelov. 2021. Offboard 3D Object Detection from Point Cloud Sequences. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[35] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Fredo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[36] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. 2018. Modeling Relational Data with Graph Convolutional Networks. In *The Extended Semantic Web Conference (ESWC)*.

[37] Haichen Shen, Jared Roesch, Zhi Chen, Wei Chen, Yong Wu, Mu Li, Vin Sharma, Zachary Tatlock, and Yida Wang. 2021. Nimble: Efficiently Compiling Dynamic Neural Networks for Model Inference. In *Conference on Machine Learning and Systems (MLSys)*.

[38] Pei Sun, Henrik Kretzschmar, Xerxes Dotiwalla, Aurelien Chouard, Vijaysai Patnaik, Paul Tsui, James Guo, Yin Zhou, Yuning Chai, Benjamin Caine, Vijay Vasudevan, Wei Han, Jiquan Ngiam, Hang Zhao, Aleksei Timofeev, Scott Ettinger, Maxim Krivokon, Amy Gao, Aditya Joshi, Yu Zhang, Jonathon Shlens, Zhifeng Chen, and Dragomir Anguelov. 2020. Scalability in Perception for Autonomous Driving: Waymo Open Dataset. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[39] Pei Sun, Mingxing Tan, Weiyue Wang, Chenxi Liu, Fei Xia, Zhaoqi Leng, and Dragomir Anguelov. 2022. SWFormer: Sparse Window Transformer for 3D Object Detection in Point Clouds. In *European Conference on Computer Vision (ECCV)*.

[40] Haotian Tang, Zhijian Liu, Xiuyu Li, Yujun Lin, and Song Han. 2022. TorchSparse: Efficient Point Cloud Inference Engine. In *Conference on Machine Learning and Systems (MLSys)*.

[41] Haotian Tang, Zhijian Liu, Shengyu Zhao, Yujun Lin, Ji Lin, Hanrui Wang, and Song Han. 2020. Searching Efficient 3D Architectures with Sparse Point-Voxel Convolution. In *European Conference on Computer Vision (ECCV)*.

[42] Keyu Tian, Yi Jiang, Qishuai Diao, Chen Lin, Liwei Wang, and Zehuan Yuan. 2023. Designing BERT for Convolutional Networks: Sparse and Hierarchical Masked Modeling. In *International Conference on Learning Representations (ICLR)*.

[43] Haiyang Wang, Chen Shi, Shaoshuai Shi, Meng Lei, Sen Wang, Di He, Bernet Schiele, and Liwei Wang. 2023. DSVT: Dynamic Sparse Voxel Transformer with Rotated Sets. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[44] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Lin, and Zheng Zhang. 2019. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *arXiv preprint arXiv:1909.01315* (2019).

[45] Yang Wang, Chen Zhang, Zhiqiang Xie, Cong Guo, Yunxin Liu, and Jingwen Leng. 2021. Dual-Side Sparse Tensor Core. In *IEEE/ACM International Symposium on Computer Architecture (ISCA)*.

[46] Zhiqiang Xie, Minjie Wang, Zihao Ye, Zheng Zhang, and Rui Fan. 2022. Graphiler: Optimizing Graph Neural Networks with Message Passing Data Flow Graph. In *Conference on Machine Learning and Systems (MLSys)*.

[47] Jiarong Xing, Leyuan Wang, Shang Zhang, Jack Chen, Ang Chen, and Yibo Zhu. 2022. Bolt: Bridging the Gap between Auto-tuners and Hardware-native Performance. In *Conference on Machine Learning and Systems (MLSys)*.

[48] Yan Yan. 2023. CUMM: CUda Matrix Multiply library. https://github.com/FindDefinition/cumm.

[49] Yan Yan. 2023. SpConv v2.3.5. https://github.com/traveller59/spconv.

[50] Yan Yan, Yuxing Mao, and Bo Li. 2018. SECOND: Sparsely Embedded Convolutional Detection. *Sensors* (2018).

[51] Dongqiangzi Ye, Weijia Chen, Zixiang Zhou, Yufei Xie, Yu Wang, Panqu Wang, and Hassan Foroosh. 2022. LidarMultiNet: Unifying LiDAR Semantic Segmentation, 3D Object Detection, and Panoptic Segmentation in a Single Multi-task Network. *arXiv preprint arXiv:2206.11428* (2022).

[52] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. 2023. SparseTIR: Composable Abstractions for Sparse Compilation in Deep Learning. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[53] Tianwei Yin, Xingyi Zhou, and Philipp Krähenbühl. 2021. Center-based 3D Object Detection and Tracking. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[54] Ziyu Ying, Shulin Zhao, Sandeepa Bhuyan, Cyan Subhra Mishra, Mahmut T Kandemir, and Chita R Das. 2022. Pushing Point Cloud Compression to the Edge. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[55] Jie-Fang Zhang and Zhengya Zhang. 2021. Point-X: A Spatial-Locality-Aware Architecture for Energy-Efficient Graph-based Point-Cloud Deep Learning. In *IEEE/ACM International Symposium on Microarchitecture*.

[56] Zhekai Zhang*, Hanrui Wang*, Song Han, and William J Dally. 2020. SpArch: Efficient Architecture for Sparse Matrix Multiplication. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[57] Bojian Zheng, Ziheng Jiang, Cody Hao Yu, Haichen Shen, Joshua Fromm, Yizhi Liu, Yida Wang, Luis Ceze, Tianqi Chen, and Gennady Pekhimenko. 2022. DietCode: Automatic Optimization for Dynamic Tensor Programs. In *Conference on Machine Learning and Systems (MLSys)*.

[58] Zixiang Zhou, Xiangchen Zhao, Yu Wang, Panqu Wang, and Hassan Foroosh. 2022. CenterFormer: Center-based Transformer for 3D Object Detection. In *European Conference on Computer Vision (ECCV)*.

[59] Chaoyang Zhu, Kejie Huang, Shuyuan Yang, Ziqi Zhu, Hejia Zhang, and Haibin Shen. 2020. An Efficient Hardware Accelerator for Structured Sparse Convolutional Neural Networks on FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28, 9 (2020), 1953–1965.