PointCNN++: Performant Convolution on Native Points

Supplementary Material

This supplementary material provides additional details, benchmarks, and implementation specifics to support the claims made in the main paper. In Sec. A, we expand on the performance benchmarks by including the fVDB baseline, presenting detailed operator-level and end-to-end comparisons, and culminating in a scalability analysis that stress-tests each method to its out-of-memory limit. We also provide detailed additional experiments in Sec. B to validate the effectiveness of our core design. For the point cloud registration task, Sec. C details the experimental protocols, datasets, and evaluation metrics, and presents extensive qualitative results that visually corroborate the quantitative superiority of our method. Finally, to ensure full transparency and reproducibility, Sec. D offers a deep dive into our custom GPU kernels, providing algorithmic pseudocode for our novel VVOR kernel and the complete Triton code for both the MVMR and VVOR implementations.

A. Extended Performance Analysis

This section provides additional benchmarks and details that complement the performance study in the main paper, further demonstrating the efficiency and scalability of our operator.

A.1. Full Benchmark Results

This section expands upon the performance benchmarks presented in the main paper (Figures 3, 4, and 5 in the main paper) to provide a more comprehensive analysis. We introduce an additional state-of-the-art baseline, fVDB [9], and present a detailed breakdown of its performance relative to our method and other existing approaches. For complete transparency and reproducibility, implementation details for all benchmarked methods are provided in Table A.

Table A. Implementation details for all benchmarked methods.

Method	**Version**
KPConv	-
SPConv	v2.3.8
O-CNN	v2.2.7
MinkowskiEngine	v0.5.4
TorchSparse++	v2.1.0
fVDB	v0.3.1
PointCNN	-
Ours (degraded)	-
Ours	-

The newly generated benchmark results, which include fVDB, are presented below and offer several key insights. The memory usage comparison in Figure D clearly demonstrates the fundamental advantage of our native operator. Ours consistently establishes the lower bound on memory consumption across all input sizes, scaling gracefully and linearly. While fVDB proves to be a highly memory-efficient voxel-based method that outperforms KPConv, SPConv and O-CNN, it still consumes noticeably more memory than both our method, TorchSparse++ and MinkowskiEngine. This persistent gap highlights the inherent overhead of all voxel-based paradigms, which require auxiliary data structures for spatial indexing (like VDB trees). In contrast, our kernel's zero-intermediate-footprint design, which operates directly on native point data, fundamentally avoids this overhead.

Figure E provides a detailed, operator-level latency analysis, revealing a crucial performance difference between inference and training. In the pure inference benchmark (left), fVDB is shown to be extremely fast, rivaling the performance of TorchSparse++ and our method, which highlights the impressive optimization of modern voxel-based forward passes. However, the picture changes decisively in the back-propagation benchmark (right). Here, Ours and Ours(degraded) are the definitive latency leaders, significantly outperforming fVDB and all other baselines. This result showcases the efficiency of our co-designed VVOR kernel for gradient computation, indicating that our system is holistically optimized for the entire training loop, not just the forward pass.

The end-to-end memory benchmark in Figure F further reinforces these findings at the network level. Our method (Ours) again sets a new state-of-the-art in memory efficiency, consuming only 0.37 GB for inference and peaking at just 0.59 GB

during training. While fVDB demonstrates impressive memory management for a voxel-based method (0.82 GB for inference, 1.79 GB for training), our native point-based approach uses less than half the memory for inference and approximately one-third the memory for training. This substantial gap underscores the fundamental memory cost associated with voxelization, even in highly optimized frameworks. Furthermore, the marginal memory increase from inference to training for our method (only 0.22 GB) is remarkably small compared to fVDB's (0.97 GB). This provides concrete evidence that our co-designed MVMR and VVOR kernel are not only fast but also exceptionally memory-frugal, minimizing the storage of intermediate activations required for forward and back propagation.

The end-to-end ResNet-18 backbone benchmark in Figure G confirms that the operator-level latency advantages directly translate to overall network performance, especially during training[1]. While fVDB has a very fast inference component (49.6 ms), its backward pass takes 105.46 ms. In stark contrast, our method, while having a slightly slower inference component (60.4 ms), requires only 75.5 ms for its backward pass. This makes our total training time per iteration significantly faster than fVDB's. This superior balance demonstrates that PointCNN++ provides a more efficient and practical solution for the demanding task of model training.

A.2. Cross-GPU Performance Validation

To demonstrate that the performance advantages of our operator are not limited to a single GPU architecture, we conducted a rigorous cross-GPU performance validation. In addition to the RTX 4090 (Ada Lovelace architecture) benchmarks presented in the main paper, we tested a single convolution layer on three distinct NVIDIA data-center and professional GPUs: the A800 (Ampere), L20 (Ada Lovelace), and V100 (Volta). The latency results for both inference and training are presented in Figure H, Figure I, and Figure J. The findings are unequivocal: across every tested hardware platform, our method (Ours) consistently achieves the lowest latency for both forward and backward passes at all point cloud scales. This sustained leadership, from older architectures like Volta to the latest Ada Lovelace, provides strong evidence that the efficiency of our custom kernels is fundamental.

The performance gains are rooted in our co-designed, memory-centric computational strategy, rather than being an artifact of specific hardware features like L2 cache size or Tensor Core capabilities on a particular GPU. This demonstrates the robustness, generalizability, and broad applicability of PointCNN++ as a high-performance solution across a wide range of deployment scenarios and hardware platforms.

A.3. Scalability Limit Analysis

To quantify the memory scalability of different backbones, we conducted an end-to-end benchmark using an identical ResNet-18 architecture on point clouds of increasing size. Figure K plots the peak memory consumption for both the forward (inference) and backward (training) passes on a log-log scale. The results unequivocally demonstrate the superior memory efficiency of our native point-based approach (PointCNN++). The practical implications of these results are profound, as shown by the horizontal lines indicating common GPU memory capacities. During training (right plot), which is the most memory-intensive scenario, baseline methods hit hardware limits very quickly. In stark contrast, PointCNN++ comfortably processes point clouds up to $\sim$100M points on the same GPU, more than doubling the capacity of the next best method.

This significant scalability disparity stems from fundamental design choices. Voxel-based methods (MinkowskiEngine, SPConv, fVDB) and point-based methods (KPConv) all rely on auxiliary data structures—such as hash maps or neighborhood graphs—that incur substantial and often non-linearly growing memory overhead. Our native approach, by directly operating on sorted point data with zero intermediate memory footprint, fundamentally circumvents this bottleneck. This exceptional memory efficiency makes PointCNN++ not just a faster alternative, but often the only feasible solution for training deep models on large-scale, real-world point clouds using standard hardware.

B. Additional Experiments

This section provides additional ablation studies to empirically dissect the sources of our method's efficiency, confirming that the performance gains stem directly from our core architectural and algorithmic designs. We conducted three key ablation studies. The results, visualized in Figure A, Figure B, and Figure C, empirically validate our design choices.

1) MVMR vs. GEMM-based Lowering: The first study investigates the performance of our MVMR kernel against two conventional strategies that "lower" the sparse convolution operation into dense matrix multiplication (GEMM): the "im2col"+GEMM method described in the main paper, and an alternative GEMM+reduction approach. As illustrated in Figure A, our specialized MVMR kernel consistently outperforms both GEMM-based strategies in both latency and memory consumption as the input scale increases. This result confirms its superior scalability and efficiency.

2) Triplet Sorting Strategy: The second study, presented in Figure B, ablates our triplet sorting strategy. We compare the performance when sorting by the kernel index k (our proposed method) against alternative sorting orders. The results clearly

[1]Note that the performance of 'Ours (degraded)' presented here has been improved due to subsequent optimizations and thus differs from the results shown in the main paper.

show that sorting by k achieves the lowest latency across all scales. This is because it maximizes temporal locality and on-chip data reuse within the GPU's cache hierarchy.

3) Versatility with kNN Neighborhoods: The final ablation confirms that our MVMR kernel's efficiency is independent of the neighborhood search method. We replaced the default radius search with k-Nearest Neighbors (kNN) and, as shown in Figure C, compared our MVMR kernel against GEMM-based lowering strategies. The results demonstrate that our method preserves its significant advantages in both latency and memory saving, mirroring the performance gains seen with radius search and proving the fundamental efficiency of our kernel design.

Collectively, these results provide strong empirical evidence that our method's efficiency stems not from incidental factors, but from our systematic, hardware-aware analysis and specific algorithmic designs (MVMR and triplet sorting).

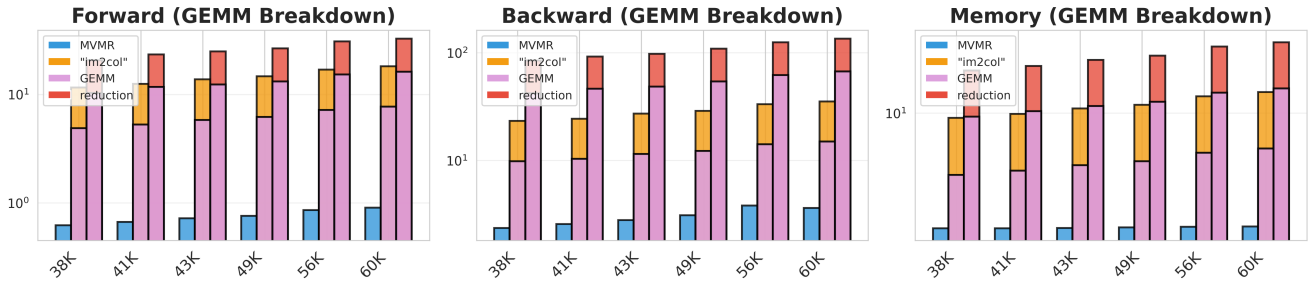

Figure A. Ablation Study on MVMR Kernel vs. GEMM-based Alternatives. Performance comparison in terms of latency and peak memory consumption across varying input scales.

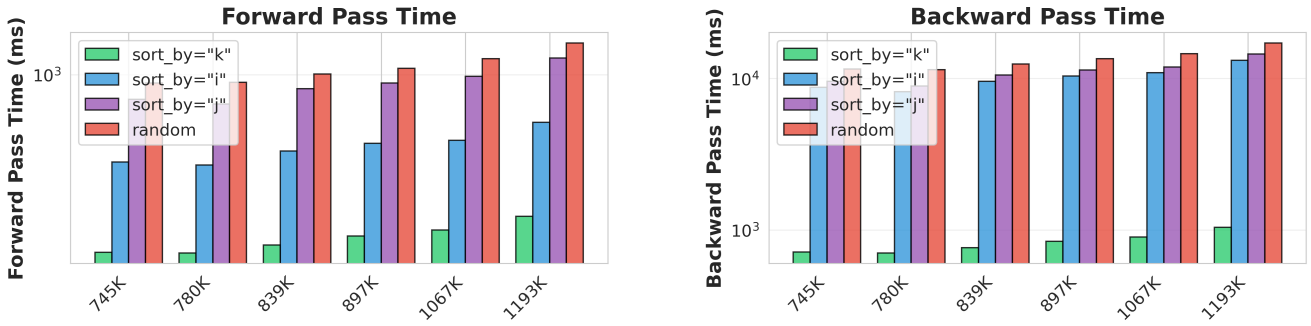

Figure B. Impact of the Triplet Sorting Strategy. Performance with our proposed sorting (by kernel index k) versus alternative sorting orders.

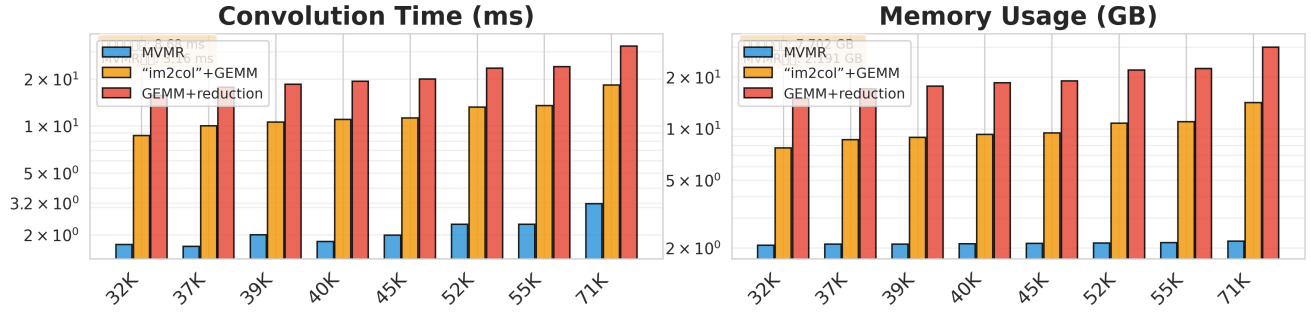

Figure C. MVMR Performance with kNN Neighborhoods. Our MVMR kernel consistently outperforms GEMM-based alternatives under a kNN setting.

C. Additional Details on Point Cloud Registration

This section provides implementation details and qualitative results for the point cloud registration task discussed in the main paper.

C.1. Experimental Setup and Metric Definitions

C.1.1. Outdoor Scene Registration: KITTI Odometry

Dataset and Protocol. We follow the setup from GeoTransformer [8] for evaluation on the KITTI Odometry dataset [3]. Sequences 00-05 are used for training, 06-07 for validation, and 08-10 for testing. Following standard practice, the ground-truth poses are refined using ICP.

Evaluation Metrics. Following prior works [4, 8], we report three standard metrics for outdoor registration:

- Relative Rotation Error (RRE): The geodesic distance between the estimated and ground-truth rotation matrices, measured in degrees ($^\circ$).
- Relative Translation Error (RTE): The Euclidean distance between the estimated and ground-truth translation vectors, measured in meters (m).
- Registration Recall (RR): The fraction of test pairs that are successfully registered, defined as satisfying both RRE $< 1^\circ$ and root-mean-square error (RMSE) below 0.2m.

Baseline Comparison. On KITTI, in addition to RANSAC-based methods like FCGF, Predator, and GeoTransformer, we also compare against recent end-to-end registration methods such as DGR [2], Reformer [7], and UMEReg [4]. Strictly speaking, a direct comparison with RANSAC-based methods is sufficient to validate the feature quality of our backbone. However, our method's performance on KITTI not only surpasses these feature-matching baselines but also exceeds that of several recent end-to-end solutions. Therefore, we include them to showcase the versatility and high potential of PointCNN++ as a powerful backbone applicable across different registration paradigms.

C.1.2. Indoor Scene Registration: 3DMatch

Dataset and Protocol. We follow the experimental setup of Predator [6] and evaluate on the 3DMatch dataset [11]. This dataset consists of 62 indoor scenes captured with RGB-D sensors, with a standard split of 46 scenes for training, 8 for validation, and 8 for testing. To ensure a fair comparison, we adhere to the same preprocessing pipeline and data splits used in the official public code of our baselines.

Evaluation Metrics. Following GeoTransformer [8], we evaluate the quality of feature matching with three metrics:

- Inlier Ratio (IR): The fraction of putative correspondences whose registration error is below a threshold ($0.1m$) under the ground-truth transformation.
- Feature Matching Recall (FMR): The fraction of point cloud pairs for which the inlier ratio is above a certain threshold (5%).
- Registration Recall (RR): The fraction of point cloud pairs that can be successfully registered, defined as achieving a correspondence root-mean-square error (RMSE) below $0.2m$ after pose estimation.

Baseline Comparison. On 3DMatch, we focus our comparison on RANSAC-based feature matching methods, including FCGF [1], CoFiNet [10], Predator [6], and GeoTransformer [8]. The performance of all methods is evaluated using the same RANSAC solver to ensure that the results are directly comparable. Unlike end-to-end regression or direct pose prediction methods, our primary goal here is to validate the improvement in feature representation brought by using PointCNN++ as a backbone. This approach isolates the feature extraction stage, ensuring that performance gains are directly attributable to the quality of the learned descriptors rather than differences in pose regression model architectures.

Details on the #Samples Hyperparameter. Following the evaluation protocol of GeoTransformer [8], we also assess robustness by varying the number of correspondences used for RANSAC (#Samples $\in \{5000, 2500, 1000, 500, 250\}$). The #Samples parameter refers to the maximum number of putative correspondences randomly sampled from the full set of matches. These samples are then fed into the RANSAC solver. This test evaluates the stability and robustness of the feature descriptors under varying correspondence densities: a large sample size represents an ideal scenario with ample matches, while a smaller sample size tests the method's performance in sparse matching conditions.

C.1.3. RANSAC Configuration

For all methods employing a RANSAC-based solver (all methods on 3DMatch, and the feature-matching baselines on KITTI), we use a unified configuration to ensure a fair comparison of the underlying features. Specifically, we run $50,000$ iterations of RANSAC to estimate the final rigid transformation. This standardized pose-solving procedure, consistent with the protocol in GeoTransformer [8], allows the final registration accuracy to directly reflect the quality of the feature descriptors provided by each backbone.

C.2. Qualitative Results on KITTI

Figure L visually corroborates our quantitative superiority on KITTI (Table 1 in the main paper). In the error visualization, our alignment is almost uniformly blue, indicating near-perfect registration. In contrast, all other methods display significant artifacts, from the widespread errors of FCGF to the scattered local misalignments of Predator and the gross errors of RegFormer. These results provides clear, intuitive evidence that our point-centric operator produces more precise correspondences, leading to a quantifiably and qualitatively superior registration that is free from the local failures plaguing other models.

C.3. Qualitative Results on 3DMatch

As shown in Figure M, the results from our method (leftmost column) demonstrate a consistently superior alignment across all examples. The transformed source clouds (yellow) overlap tightly with the target clouds (blue), showing only very few and sparse red points, which indicates a highly accurate and robust registration with minimal error. In contrast, baseline methods exhibit visibly larger discrepancies. FCGF and CoFiNet, in particular, struggle on these challenging pairs, producing significant misalignments characterized by large, dense clusters of bright red points that suggest substantial errors in the estimated poses. Even stronger baselines like GeoTransformer and Predator, while achieving more plausible global alignments, still display more noticeable and larger patches of red points compared to our method, especially along planar surfaces and detailed structures. This indicates a lower level of precision in their fine-grained alignment. Overall, these qualitative results strongly corroborate our quantitative findings (Table 2 in the main paper), visually confirming that the superior feature representation learned by the PointCNN++ backbone enables our registration pipeline to achieve a more precise and reliable alignment than existing state-of-the-art methods.

D. Kernel Implementation Details

This section delves into the low-level implementation of our custom GPU kernels, providing pseudocode and code snippets for full transparency and reproducibility.

D.1. Efficient GPU Algorithm of VVOR

The backward pass requires computing the gradient with respect to the weight kernels $\mathbf{W}$. As defined in Eqn. (5) of the main paper, this involves summing the outer products of the upstream output gradients and the corresponding input feature vectors. We abstract this operation as Vector-Vector Outer-product and Reduction (VVOR).

Following the same design principles as our forward-pass MVMR kernel, we develop a second dedicated, highly-optimized GPU kernel for VVOR. It also operates on a list of triplets $\mathcal{T}$, and following a similar reasoning as that in computing MVMR, sorting the triplets by the kernel index k significantly saves $atomicAdd$ operation for the relatively large kernel matrices, thus is preferred in typical configurations. The kernel leverages on-chip memory to accumulate the outer product results for each weight matrix $\mathbf{W}_k$, writing the final accumulated gradient matrix back to global memory only when the computation for that specific kernel is complete. This strategy minimizes costly global memory writes and achieves zero intermediate memory footprint. The algorithm for our VVOR kernel, which computes the gradient $\nabla_{\mathbf{W}}$, is detailed in Algorithm A.

D.2. Triton Code Implementation

To demonstrate the practical application of our proposed computational strategies, we provide implementations of the MVMR and VVOR kernels using Triton. Triton is a Python-based language and compiler for writing highly efficient GPU kernels. These listings map the high-level logic described in Algorithm 1 (in the main paper) and Algorithm A into Triton's programming model, showcasing how on-chip accumulation and optimized memory access patterns are realized in practice.

Listing 1 presents the implementation of the MVMR kernel (sparse_matrix_vector_multiplication_reduction_kernel). As detailed in the main paper, this single, highly-optimized kernel is versatile, executing not only the forward pass but also the computation of input feature gradients $\nabla\mathbf{F}^{\text{in}}$ during the backward pass. This reuse is highly efficient as both operations reduce to the same MVMR computational pattern. The kernel processes a block of triplets, performs the corresponding matrix-vector products, and accumulates the results for each output point before writing them back to global memory via atomic operations.

Due to the reason that the kernel is used for both the forward and backward pass, thus the naming of the variables are not aligned with the notations in the forward pass nor those in the backward pass. Essentially, variable a and b indicate the inputs, while o is the output, and variables end with $\_idx$ are the corresponding index. The variables T, M, and C corresponding the $|\mathcal{T}|$, C_{out} and C_{in}, respectively. Our code supports the widely used group convolution, and the variable G is the number of groups. Such naming scheme is used in all of our code.

Listing 2 shows the implementation of the VVOR kernel (sparse_vector_vector_outer_product_reduction_kernel), used for computing weight gradients in the backward pass. Here, the core operation is the outer product between the upstream gradient vector and the input feature vector. The results are accumulated on-chip into a gradient matrix, which is then atomically added to the corresponding global weight gradient matrix.

Algorithm A VVOR Kernel for Computing Gradients $\nabla \mathbf{W}$.

Inputs: Upstream gradients $\nabla \mathbf{F}^{\text{out}}$, $\mathbf{F}^{\text{in}}$, $\mathcal{T}^L$.
Output: Weight gradients $\nabla \mathbf{W}$.

$(\tilde{i}, \tilde{j}, \tilde{k}) \leftarrow \mathcal{T}_0^L$ ▷ initialize with the first triplet
$\tilde{\mathbf{g}}_{\text{out}} \leftarrow \nabla \mathbf{F}^{\text{out}}_{\tilde{i}}$, $\tilde{\mathbf{f}}_{\text{in}} \leftarrow \mathbf{F}^{\text{in}}_{\tilde{j}}$ ▷ read from global memory
$\nabla_{\tilde{\mathbf{W}}} \leftarrow \tilde{\mathbf{g}}_{\text{out}} \otimes \tilde{\mathbf{f}}_{\text{in}}$ ▷ fast on-chip computation

for all (i, j, k) in $\mathcal{T}^L_{[1,2,\ldots,L-1]}$ **do**
  if $i \neq \tilde{i}$ **then**
    $\tilde{i} \leftarrow i$, $\tilde{\mathbf{g}}_{\text{out}} \leftarrow \nabla \mathbf{F}^{\text{out}}_{i}$ ▷ read only if necessary
  end if
  if $j \neq \tilde{j}$ **then**
    $\tilde{j} \leftarrow j$, $\tilde{\mathbf{f}}_{\text{in}} \leftarrow \mathbf{F}^{\text{in}}_{j}$ ▷ read only if necessary
  end if

  if $k \neq \tilde{k}$ **then**
    atomicAdd$(\nabla \mathbf{W}_{\tilde{k}}, \nabla_{\tilde{\mathbf{W}}})$ ▷ only if necessary
    $\tilde{k} \leftarrow k$, $\nabla_{\tilde{\mathbf{W}}} \leftarrow \tilde{\mathbf{g}}_{\text{out}} \otimes \tilde{\mathbf{f}}_{\text{in}}$ ▷ on-chip, fast
  else
    $\nabla_{\tilde{\mathbf{W}}} \leftarrow \nabla_{\tilde{\mathbf{W}}} + \tilde{\mathbf{g}}_{\text{out}} \otimes \tilde{\mathbf{f}}_{\text{in}}$ ▷ on-chip, fast
  end if
end for

atomicAdd$(\nabla \mathbf{W}_{\tilde{k}}, \nabla_{\tilde{\mathbf{W}}})$

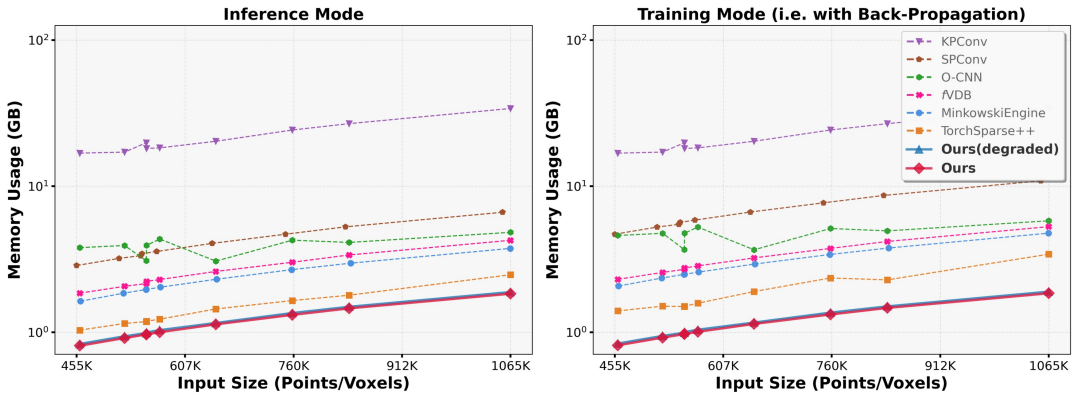

Figure D. Expanded memory usage comparison of one convolution layer.

Note that while sorting triplets by k is typically the optimal choice, the support for sorting by i or j is also provided in our algorithm and code, as they share very similar logic. The PyTorch code for invoking the Triton kernels is listed in 3 for reference.

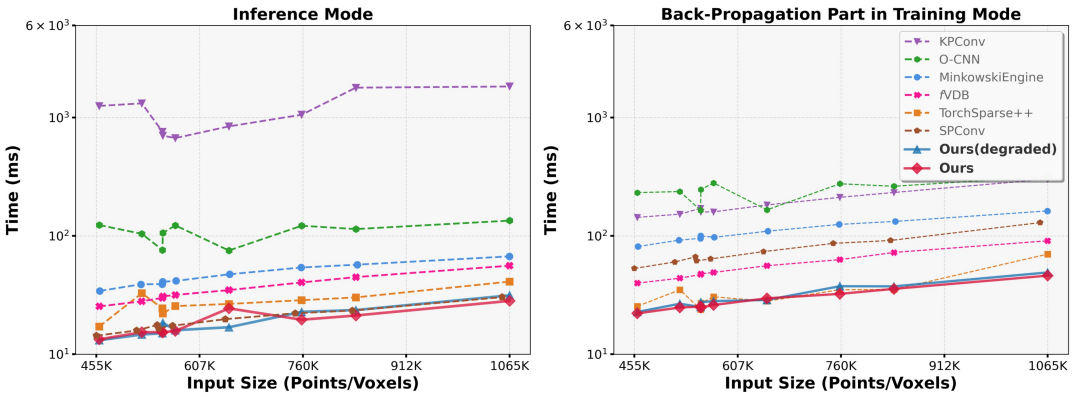

Figure E. Expanded performance comparison of one convolution layer.

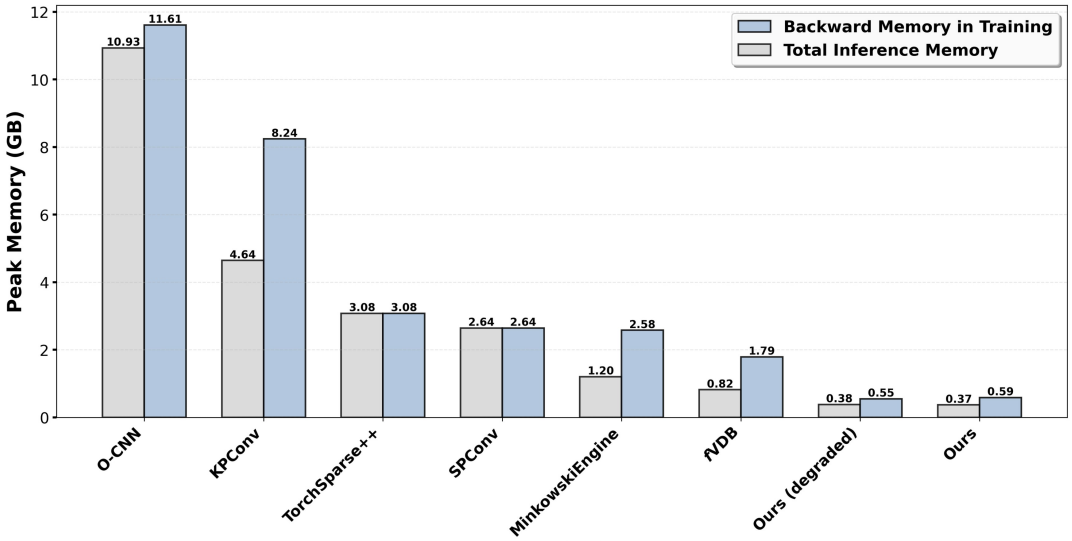

Figure F. Expanded memory usage comparison of various convolutional backbones for 3D learning assembled in an identical ResNet-18 [5] architecture.

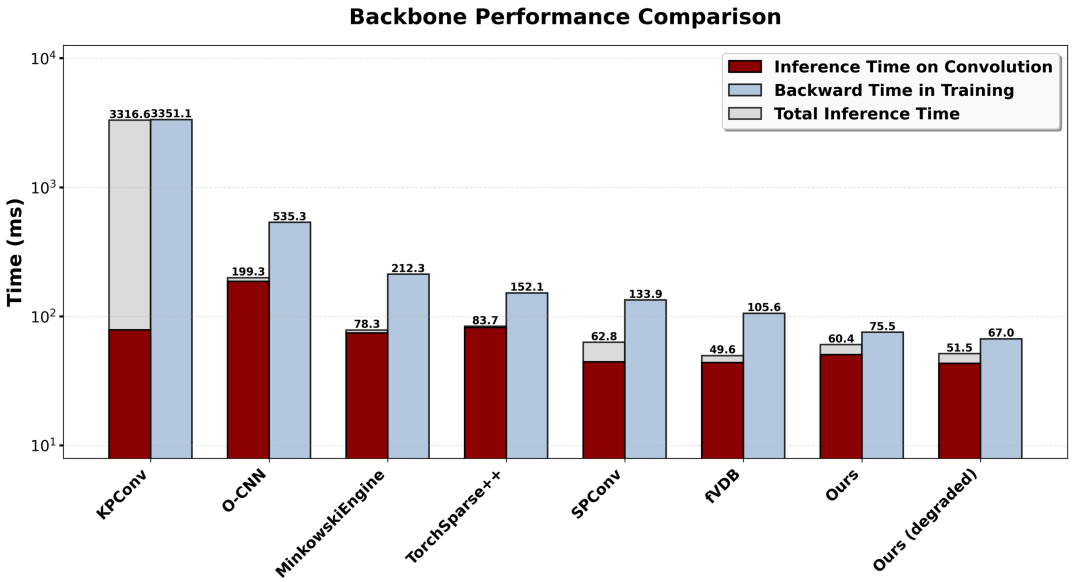

Figure G. Expanded performance of various convolutional backbones for 3D learning assembled in an identical ResNet-18 [5] architecture.

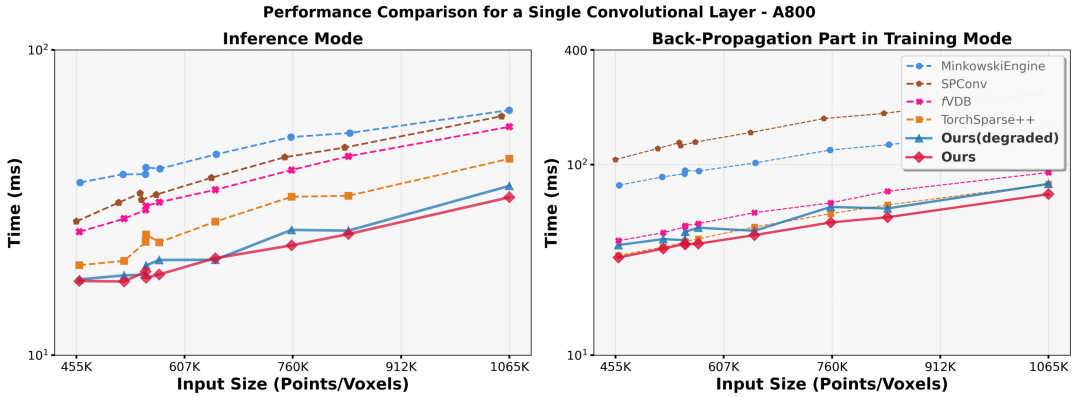

Figure H. Performance comparison of one convolution layer on A800.

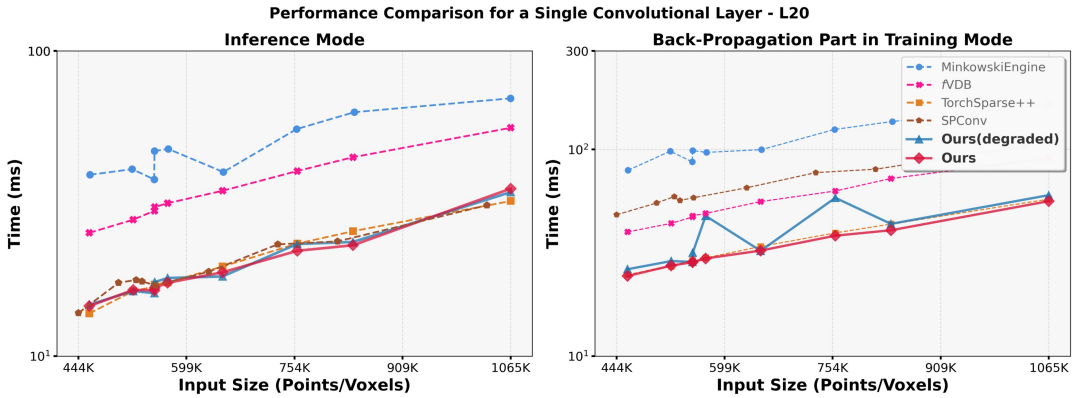

Figure I. Performance comparison of one convolution layer on L20.

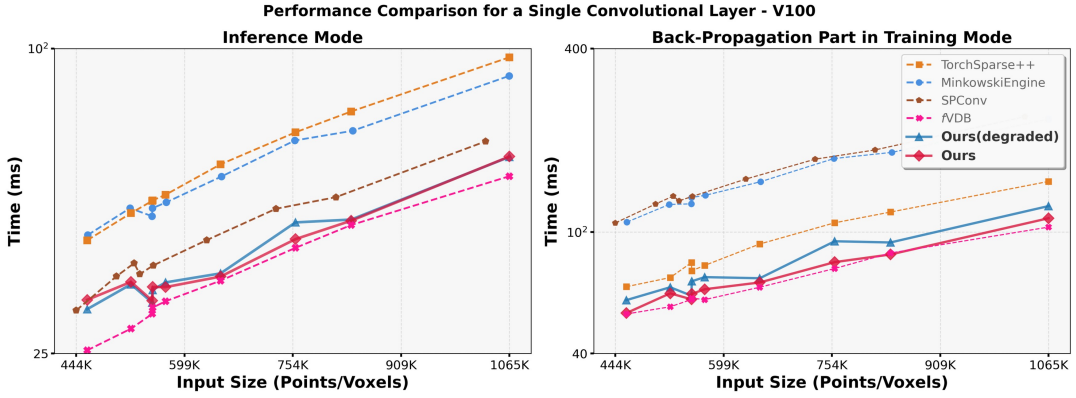

Figure J. Performance comparison of one convolution layer on V100.

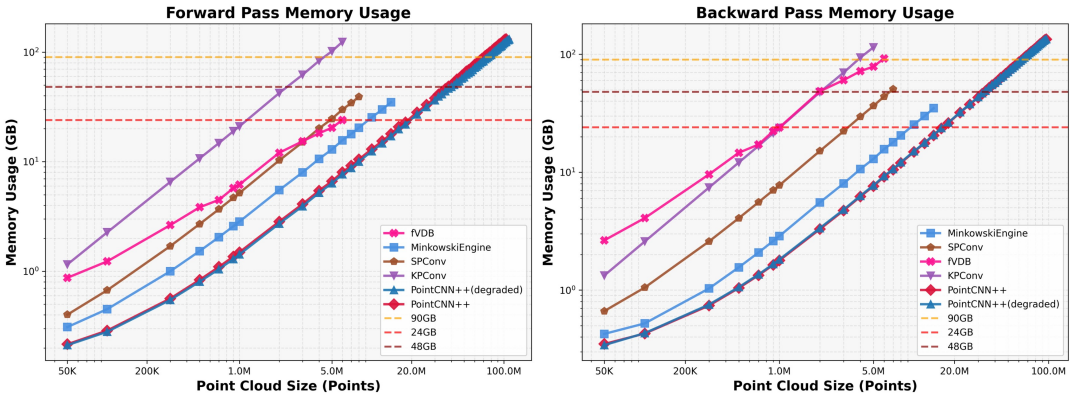

Figure K. Peak memory consumption of various ResNet-18 [5] backbones on point clouds of increasing size.

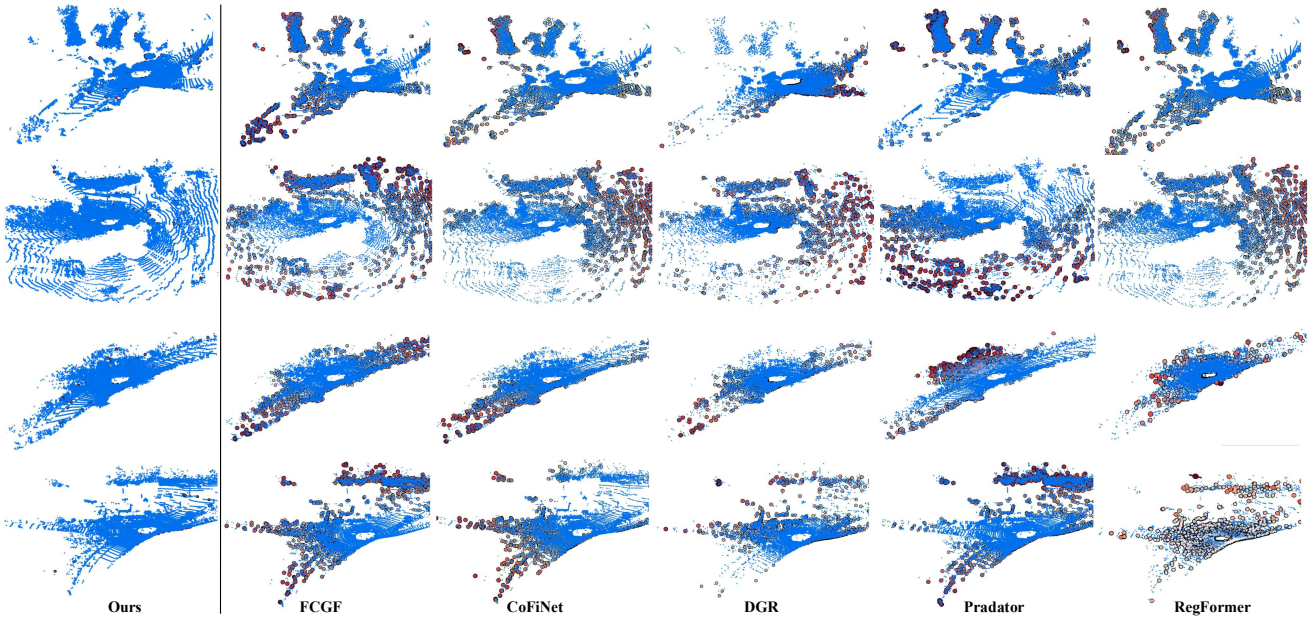

Figure L. Qualitative comparison on the KITTI [3] dataset. We visualize the per-point alignment error after registration, where the color indicates the error magnitude from blue (low) to red (high).

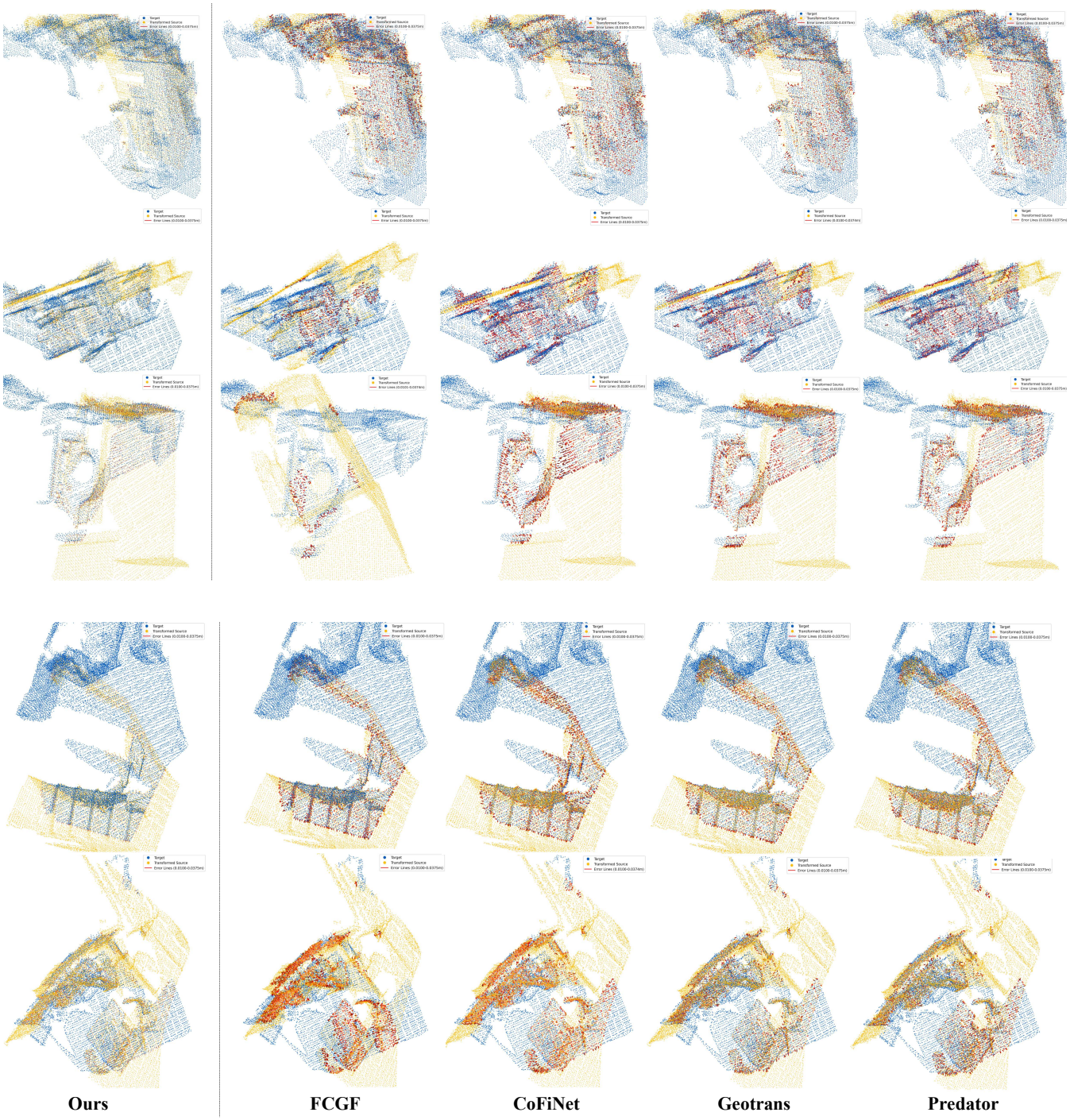

Figure M. Qualitative comparison on the 3DMatch [11] dataset.

Listing 1. Triton implementation of MVMR kernel.

```
@triton.autotune(
    configs=[
        triton.Config(
            {
                "L": 128,
                "BLOCK_SIZE_G": 1,
                "BLOCK_SIZE_M": 32,
                "BLOCK_SIZE_C": 32,
            },
            num_warps=1,
        ),
    ],
    key=["T", "G", "M", "C"],
)
@triton.jit
def sparse_matrix_vector_multiplication_reduction_kernel(
        a, a_idx, b, b_idx, o, o_idx, T, G, M, C,
        L: tl.constexpr, BLOCK_SIZE_G: tl.constexpr,
        BLOCK_SIZE_M: tl.constexpr, BLOCK_SIZE_C: tl.constexpr,
):
    num_pid_g = tl.cdiv(G, BLOCK_SIZE_G)
    num_pid_m = tl.cdiv(M, BLOCK_SIZE_M)
    num_pid_c = tl.cdiv(C, BLOCK_SIZE_C)

    pid = tl.program_id(axis=0)
    pid_m = pid % num_pid_m
    pid //= num_pid_m
    pid_c = pid % num_pid_c
    pid //= num_pid_c
    pid_g = pid % num_pid_g
    pid_t = pid // num_pid_g

    g_offsets = pid_g * BLOCK_SIZE_G + tl.arange(0, BLOCK_SIZE_G)
    m_offsets = pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)
    c_offsets = pid_c * BLOCK_SIZE_C + tl.arange(0, BLOCK_SIZE_C)

    g_mask = g_offsets < G
    m_mask = m_offsets < M
    c_mask = c_offsets < C

    gm_mask = g_mask[:, None] & m_mask[None, :]
    gc_mask = g_mask[:, None] & c_mask[None, :]
    gcm_mask = g_mask[:, None, None] & c_mask[None, :, None] & m_mask[None, None, :]

    a_ptrs = a + (
            g_offsets[:, None, None] * (C * M)
            + c_offsets[None, :, None] * M
            + m_offsets[None, None, :]
    )
    b_ptrs = b + (g_offsets[:, None] * C + c_offsets[None, :])
    o_ptrs = o + (g_offsets[:, None] * M + m_offsets[None, :])

    t_offset = pid_t * L
    a_offset = tl.load(a_idx + t_offset)
    b_offset = tl.load(b_idx + t_offset)
    o_offset = tl.load(o_idx + t_offset)

    block_a = tl.load(a_ptrs + a_offset * (G * C * M), mask=gcm_mask)
    block_b = tl.load(b_ptrs + b_offset * (G * C), mask=gc_mask)
    block_o = tl.sum(block_a * block_b[:, :, None], axis=1)
    for t in tl.range(1, min(L, T - t_offset)):
        a_offset_next = tl.load(a_idx + t_offset + t)
        b_offset_next = tl.load(b_idx + t_offset + t)
        o_offset_next = tl.load(o_idx + t_offset + t)

        if a_offset_next != a_offset:
            block_a = tl.load(a_ptrs + a_offset_next * (G * C * M), mask=gcm_mask)
            a_offset = a_offset_next

        if b_offset_next != b_offset:
            block_b = tl.load(b_ptrs + b_offset_next * (G * C), mask=gc_mask)
            b_offset = b_offset_next

        if o_offset_next != o_offset:
            tl.atomic_add(o_ptrs + o_offset * (G * M), block_o, mask=gm_mask)
            o_offset = o_offset_next
            block_o = tl.sum(block_a * block_b[:, :, None], axis=1)
        else:
            block_o += tl.sum(block_a * block_b[:, :, None], axis=1)
    tl.atomic_add(o_ptrs + o_offset * (G * M), block_o, mask=gm_mask)
```

Listing 2. Triton implementation of VVOR kernel.

```
@triton.autotune(
    configs=[
        triton.Config(
            {
                "L": 128,
                "BLOCK_SIZE_G": 1,
                "BLOCK_SIZE_M": 32,
                "BLOCK_SIZE_C": 32,
            },
            num_warps=1,
        ),
    ],
    key=["T", "G", "M", "C"],
)
@triton.jit
def sparse_vector_vector_outer_product_reduction_kernel(
        a, a_idx, b, b_idx, o, o_idx, T, G, M, C,
        L: tl.constexpr, BLOCK_SIZE_G: tl.constexpr,
        BLOCK_SIZE_M: tl.constexpr, BLOCK_SIZE_C: tl.constexpr,
):
    num_pid_g = tl.cdiv(G, BLOCK_SIZE_G)
    num_pid_m = tl.cdiv(M, BLOCK_SIZE_M)
    num_pid_c = tl.cdiv(C, BLOCK_SIZE_C)

    pid = tl.program_id(axis=0)
    pid_c = pid % num_pid_c
    pid //= num_pid_c
    pid_m = pid % num_pid_m
    pid //= num_pid_m
    pid_g = pid % num_pid_g
    pid_t = pid // num_pid_g

    g_offsets = pid_g * BLOCK_SIZE_G + tl.arange(0, BLOCK_SIZE_G)
    m_offsets = pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)
    c_offsets = pid_c * BLOCK_SIZE_C + tl.arange(0, BLOCK_SIZE_C)

    g_mask = g_offsets < G
    m_mask = m_offsets < M
    c_mask = c_offsets < C

    gm_mask = g_mask[:, None] & m_mask[None, :]
    gc_mask = g_mask[:, None] & c_mask[None, :]
    gmc_mask = g_mask[:, None, None] & m_mask[None, :, None] & c_mask[None, None, :]

    a_ptrs = a + (g_offsets[:, None] * M + m_offsets[None, :])
    b_ptrs = b + (g_offsets[:, None] * C + c_offsets[None, :])
    o_ptrs = o + (
            g_offsets[:, None, None] * (M * C)
            + m_offsets[None, :, None] * C
            + c_offsets[None, None, :]
    )

    t_offset = pid_t * L
    a_offset = tl.load(a_idx + t_offset)
    b_offset = tl.load(b_idx + t_offset)
    o_offset = tl.load(o_idx + t_offset)

    block_a = tl.load(a_ptrs + a_offset * (G * M), mask=gm_mask)
    block_b = tl.load(b_ptrs + b_offset * (G * C), mask=gc_mask)
    block_o = block_a[:, :, None] * block_b[:, None, :]
    for t in tl.range(1, min(L, T - t_offset)):
        a_offset_next = tl.load(a_idx + t_offset + t)
        b_offset_next = tl.load(b_idx + t_offset + t)
        o_offset_next = tl.load(o_idx + t_offset + t)

        if a_offset_next != a_offset:
            block_a = tl.load(a_ptrs + a_offset_next * (G * M), mask=gm_mask)
            a_offset = a_offset_next

        if b_offset_next != b_offset:
            block_b = tl.load(b_ptrs + b_offset_next * (G * C), mask=gc_mask)
            b_offset = b_offset_next

        if o_offset_next != o_offset:
            tl.atomic_add(o_ptrs + o_offset * (G * M * C), block_o, mask=gmc_mask)
            o_offset = o_offset_next
            block_o = block_a[:, :, None] * block_b[:, None, :]
        else:
            block_o = tl.fma(block_a[:, :, None], block_b[:, None, :], block_o)
    tl.atomic_add(o_ptrs + o_offset * (G * M * C), block_o, mask=gmc_mask)
```

Listing 3. PyTorch code for invoking Triton MVMR and VVOR kernels.

```
def sparse_matrix_vector_multiplication_reduction(a, a_idx, b, b_idx, o_idx, n_o):
    assert a.is_cuda
    assert a_idx.is_cuda
    assert b.is_cuda
    assert b_idx.is_cuda
    assert o_idx.is_cuda

    a = a.contiguous()
    b = b.contiguous()

    T, G, M, C = a_idx.numel(), a.shape[1], a.shape[3], a.shape[2]
    o = torch.zeros((n_o, G, M), dtype=a.dtype, device=a.device)

    grid = lambda META: (
        triton.cdiv(T, META["L"])
        * triton.cdiv(G, META["BLOCK_SIZE_G"])
        * triton.cdiv(M, META["BLOCK_SIZE_M"])
        * triton.cdiv(C, META["BLOCK_SIZE_C"]),
    )
    sparse_matrix_vector_multiplication_reduction_kernel[grid](a, a_idx, b, b_idx, o, o_idx, T, G, M, C)

    return o

def sparse_vector_vector_outer_product_reduction(a, a_idx, b, b_idx, o_idx, n_o):
    assert a.is_cuda
    assert a_idx.is_cuda
    assert b.is_cuda
    assert b_idx.is_cuda
    assert o_idx.is_cuda

    a = a.contiguous()
    b = b.contiguous()

    T, G, M, C = a_idx.numel(), a.shape[1], a.shape[2], b.shape[2]
    o = torch.zeros((n_o, G, M, C), dtype=a.dtype, device=a.device)

    grid = lambda META: (
        triton.cdiv(T, META["L"])
        * triton.cdiv(G, META["BLOCK_SIZE_G"])
        * triton.cdiv(M, META["BLOCK_SIZE_M"])
        * triton.cdiv(C, META["BLOCK_SIZE_C"]),
    )
    sparse_vector_vector_outer_product_reduction_kernel[grid](a, a_idx, b, b_idx, o, o_idx, T, G, M, C)

    return o
```

References

[1] Christopher Choy, Jaesik Park, and Vladlen Koltun. Fully convolutional geometric features. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 8958–8966, 2019.

[2] Christopher Choy, Wei Dong, and Vladlen Koltun. Deep global registration. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020.

[3] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *2012 IEEE conference on computer vision and pattern recognition*, 2012.

[4] Yuval Haitman, Amit Efraim, and Joseph M Francos. Umeregrobust-universal manifold embedding compatible features for robust point cloud registration. In *European Conference on Computer Vision*, 2024.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

[6] Shengyu Huang, Zan Gojcic, Mikhail Usvyatsov, Andreas Wieser, and Konrad Schindler. Predator: Registration of 3d point clouds with low overlap. In *Proceedings of the IEEE/CVF Conference on computer vision and pattern recognition*, 2021.

[7] Jiuming Liu, Guangming Wang, Zhe Liu, Chaokang Jiang, Marc Pollefeys, and Hesheng Wang. Regformer: An efficient projection-aware transformer network for large-scale point cloud registration. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2023.

[8] Zheng Qin, Hao Yu, Changjian Wang, Yulan Guo, Yuxing Peng, Slobodan Ilic, Dewen Hu, and Kai Xu. Geotransformer: Fast and robust point cloud registration with geometric transformer. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2023.

[9] Francis Williams, Jiahui Huang, Jonathan Swartz, Gergely Klar, Vijay Thakkar, Matthew Cong, Xuanchi Ren, Ruilong Li, Clement Fuji-Tsang, Sanja Fidler, et al. fvdb: A deep-learning framework for sparse, large scale, and high performance spatial intelligence. *ACM Transactions on Graphics (TOG)*, 2024.

[10] Hao Yu, Fu Li, Mahdi Saleh, Benjamin Busam, and Slobodan Ilic. Cofinet: Reliable coarse-to-fine correspondences for robust point-cloud registration. *Advances in Neural Information Processing Systems*, 2021.

[11] Andy Zeng, Shuran Song, Matthias Nießner, Matthew Fisher, Jianxiong Xiao, and Thomas Funkhouser. 3dmatch: Learning local geometric descriptors from rgb-d reconstructions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017.