Flash3D: Super-scaling Point Transformers through Joint Hardware-Geometry Locality

Supplementary Material

7. Outdoor 3D Tasks

In this section, we provide more benchmark results on semantic segmentation tasks and detection tasks, including nuScenes [2, 10] and Waymo open dataset [35]. We also provide ablation results by varying the attention scope sizes.

7.1. nuScenes Semantic Segmentation

We present the benchmarks on nuScenes semantic segmentation for both validation set and test set. Similar to Section 5, we train two variants of Flash3D by fixing it under the same number of parameters and the same total memory costs. We present the results in Table 4. Our results show that Flash3D outperforms previous state-of-the-art results on both validation set and test set. When we increase the memory budgets for Flash3D to include more parameters, Flash3D further improves the mIoU performance.

Table 4. Outdoor semantic segmentation on nuScenes validation and test sets.

Methods	nuScenes Val	nuScenes Test
MinkUNet [4]	73.3	-
SPVNAS [37]	77.4	-
Cylinder3D [47]	76.1	77.2
AF2S3Net [3]	62.2	78.0
SphereFormer [18]	78.4	81.9
PTv2 [40]	80.2	82.6
PTv3 [41]	80.4	82.7
Flash3D (Same param.)	81.2	83.1
Flash3D (Same memory)	81.5	83.6

7.2. Waymo Semantic Segmentation

Waymo Validation Set We benchmark two variants of Flash3D on Waymo Open Dataset semantic segmentation task in Table 5. Flash3D consistently outperforms previous state-of-the-art results on both mIoU and mAcc metrics. Flash3D demonstrates further scaling when we add more parameters.

Scaling up Attention Scopes On nuScenes semantic segmentation task, we scale attention scope sizes at 1024, 4096, and 8192 to benchmark PTv3 [41] and two variants of Flash3D in Table 6.

PTv3 performance *degrades* when we scale up the attention scope sizes. When we fix the parameter size, Flash3D performance peaks at 4096. And increasing attention scopes to 8192 degrades the task performance.

When we fix the memory budgets and add more parameters to Flash3D, increasing attention scope sizes improves the task performance.

Table 5. Waymo Val mIoU and mAcc comparison.

Methods	mIoU	mAcc
MinkUNet [4]	65.9	76.6
SphereFormer [18]	69.9	-
PTv2 [40]	70.6	80.2
PTv3 [41]	71.3	80.5
Flash3D (Same param.)	71.7	80.9
Flash3D (Same memory)	72.5	81.6

Table 6. Attention Scope impacts on nuScenes semantic segmentation on the validation set

Attention Scope	1024	4096	8192
PTv3 [41]	80.4	80.2	79.1
Flash3D (Same param.)	80.6	81.2	80.1
Flash3D (Same memory)	80.2	81.5	81.7

We observe a similar trend when scaling up the attention scopes on Waymo validation set. The improvements of scaling up attention scopes of Flash3D are more pronounced on Waymo validation set since Waymo has denser point clouds and more data.

Table 7. Attention Scope impacts on Waymo semantic segmentation on the validation set mIoU.

Attention Scope	1024	4096	8192
PTv3 [41]	70.8	70.5	70.2
Flash3D (Same param.)	71.5	71.7	71.3
Flash3D (Same memory)	71.6	72.1	72.5

7.3. Waymo Detection Task

We benchmark Flash3D on Waymo detection task for a single-frame setting in Table 8. We show that Flash3D outperform both PTv3 and SST [9] methods on detection tasks by a notable margin due to our flexible attention scope shaping.

Table 8. Waymo Object Detection on a single frame

Waymo Obj. Det.	SST	PTv3	Flash3D (Same param.)
Mean L2 mAPH	64.8	70.5	**71.6**

8. Hardware Scalability

8.1. H100 Training and Inference Costs

We present training and inference costs of Flash3D. Similar to Section 5, we train two variants of Flash3D: one with the same parameter sizes as PTv3 [41], and one with the same inference memory costs as PTv3 [41]. We present training latencies and inference latencies on H100 in Table 9. For fair comparisons, we report training latencies as the time to train an iteration when batch size is 1.

Scalability (nuScenes)	Params.	H100 Memory	H100 Inf Latency	H100 Train Latency	mIoU
PTv3 [41]	46.2M	1.2G	30.2ms	77.6ms	80.4
Flash3D	46.2M	0.5G	**13.4ms**	**33.8ms**	**81.2**
PTv3 [41]	46.2M	1.2G	30.2ms	77.6ms	80.4
Flash3D	129.4M	1.2G	**15.1ms**	**37.9ms**	**81.5**

Table 9. Model scalability comparisons of PTv3 and Flash3D on H100 GPUs by fixing model parameter sizes and memory quotas respectively. Dark cells indicate fixed budgets. We fix attention scopes of all Flash3D models at 4096. mIoU indicates the semantic segmentation performance on nuScenes validation set.

8.2. H100 Utilization Profiling

In Section 5, we describe key metrics and evaluate hardware scalability of Flash3D on **General Compute**, **TensorCore Matrix Multiplication**, and **Memory Bandwidth**. In this section, we further elaborate on the implications of the metrics and present profiling results on H100 GPUs. We keep the same setting as Section 5 to fix PTv3 and Flash3D at the same parameter sizes for profiling results. Generally speaking, H100s are more memory bandwidth-starved and Flash3D shows further improvements over PTv3 [41] on key metrics due to proper treatment of *memory locality*.

General Compute SM cores are the *computing tiles* for GPUs and H100 has 132 SM cores[3]. The aggregated SM utilization is a key measure of overall usage. For more detailed illustrations of GPU architectures and hierarchies, refer to FlashAttention [7] and ThunderKittens [34].

SM issuing rates focus more on the instruction front-end utilization. A low SM issuing rate indicates the workloads are not properly divided among threads or the computing units are waiting for data. Flash3D carefully spreads workloads among threads and SMs and demonstrates significant improvements on SM issuing rates in Figure 12.

[3]Different versions of H100 offer options of 144 SMs, 132 SMs, and 113 SMs.

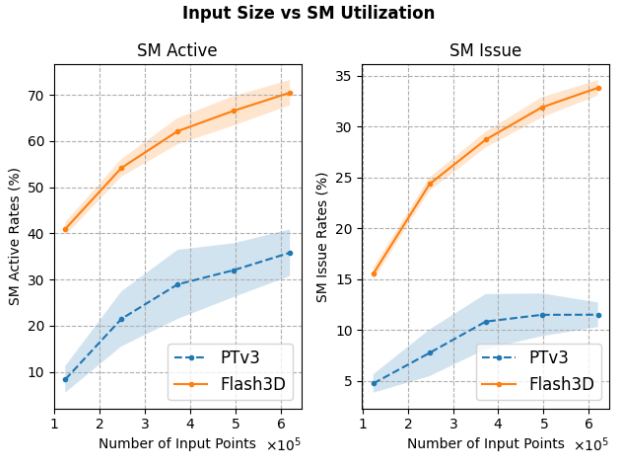

Figure 12. SM Utilization vs. Input Sizes for Flash3D and PTv3. We show the overall SM active rates on the left and more specific SM issuing rates on the right.

TensorCore Matrix Multiplication H100 TensorCores demand more data from other components of the GPU to saturate. H100 poses harder barriers than A100 does in terms of TensorCore saturation [34]. PTv3 consistently demonstrates TensorCore utilization less than 5%, which **wastes over 95%** of GPU resources and investments as shown in Figure 14.

Memory Bandwidth H100 computing throughput disproportionally improves over its memory bandwidth [34]. Therefore, H100 demands more realized memory bandwidth to keep its computing tiles working. Flash3D carefully treats memory locality and enjoys a scalable usage of memory bandwidth, shown in Figure 15.

9. Bucket Swin Attention Scope Details

We further visualize the attention scope construction in Fig. 13. The color of each point in the top-left figure indicates which bucket it is assigned to. The other figures depict how a single attention scope changes under shifting and striding operations for each of two hash functions (Hash-ZM and Hash-ZD).

10. Implementation Details

In this section, we further elaborate on implementation details and algorithm designs from Section 4 and Section 5. Our open-source implementation is provided in the official repository at https://github.com/liyanc/Flash3DTransformer.

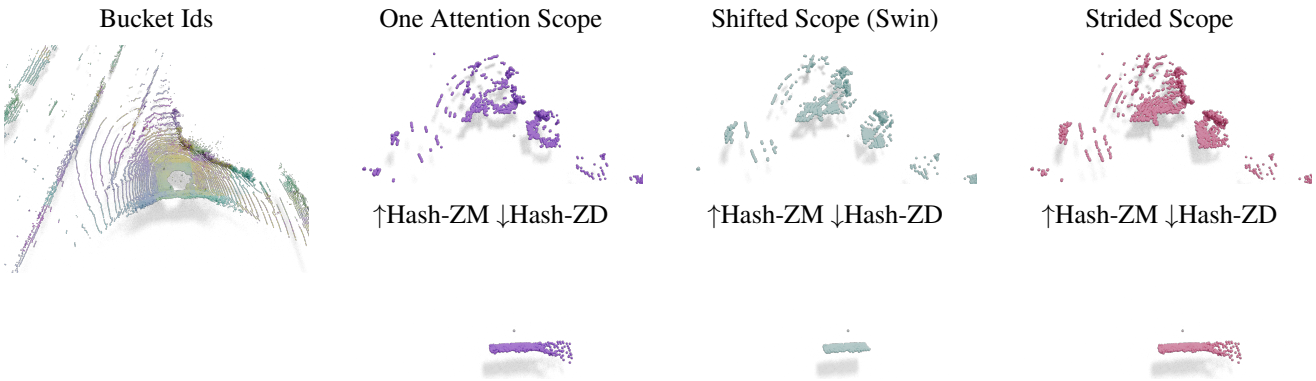

Figure 13. Illustration of bucket-based attention scopes: (left) Bucket Ids, (center-left) One Attention Scope, (center-right) Shifted Scope (Swin), and (right) Strided Scope.

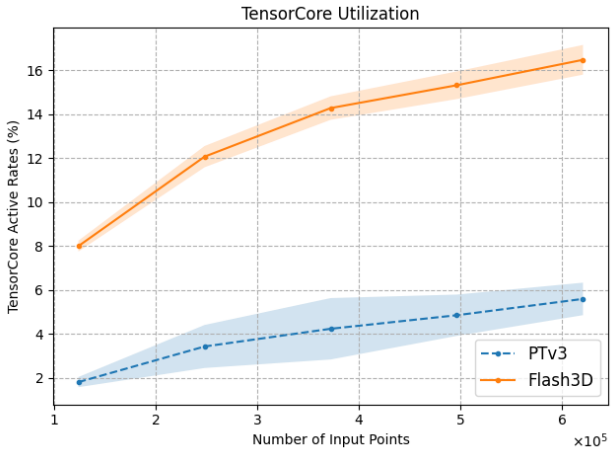

Figure 14. TensorCore Active Rates vs. Input Sizes for Flash3D and PTv3.

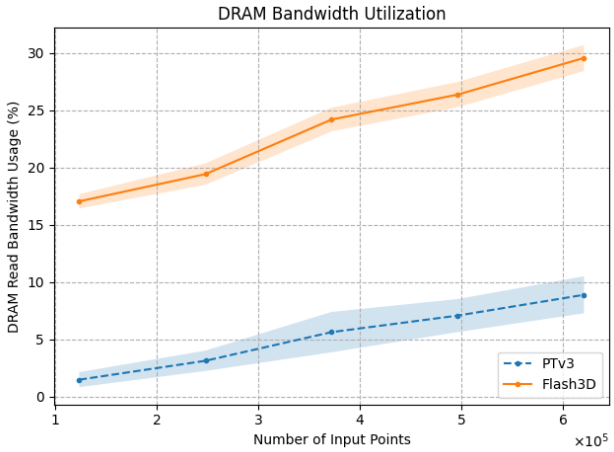

Figure 15. DRAM Read Bandwidth Usage vs. Input Sizes for Flash3D and PTv3.

10.1. PSH with Two-Stage Counters

In our main PSH algorithms, we used `AtomicInc`, `AtomicDec` on globally visible memory (DRAM), which establishes a global linear order among all threads on a GPU, and hence contiguous and unique indices for all points. We implemented our naive version of PSH algorithms as described in Section 4. The naive version demonstrated acceptable latencies, roughly at 0.4ms for 120k points.

However, our rebalancing algorithm generates a significant amount of memory traffic requiring global atomic guarantees, which throttles the overall throughput. Therefore, we instead implemented a *two-stage-counter* version of our PSH algorithms, where we temporarily break the global linear order and coalesce temporary copies in batches.

Specifically, we create temporary copies of **bkt_ctr** within local L1 cache blocks (shared memory), which allow fast localized atomics, which we will refer to as `AtomicInc_block` and `AtomicDec_block`. We usually have 256 buckets and the counters typically have a shape of `UInt32`[256], costing 1024 bytes, so they fit in local L1 cache blocks.

We break our original counter `AtomicInc` into three steps: local `AtomicInc_block`, bulk commit to global memory `AtomicAdd`, and rebasing **bucket_offset**:

Local Atomic Operations We initialize local copies **bkt_ctr_local** with zeros. When a GPU thread representing a point needs `AtomicInc`, it should `AtomicInc_block`(**bkt_ctr_local**) to obtain **bucket_off_local**.

Bulk Commit We hold all threads in a block to a block-wise synchronization point and make sure

every thread has finished their `AtomicInc_block`. Then we `AtomicAdd`(**bkt_ctr**, **bkt_ctr_local**) to obtain **bkt_rebase_offset**. Recall that **bkt_ctr_local** starts from zeros, so it represents a contiguous range of indices based on a future global version to be determined. When we `AtomicAdd`(**bkt_ctr**, **bkt_ctr_local**), we establish the starting point of the specific local range of indices in terms of the global counters. At the same moment, we obtain a snapshot of global counters before `AtomicAdd`, which are exactly the bases for this **bkt_ctr_local**. Therefore, we refer to this global counter snapshot as **bkt_rebase_offset**.

Rebasing bucket_offset Recall that we have **bucket_off_local** in the first step, which needs to be adjusted and merged to the global linear order. Similar to our discussion in the second step, **bucket_off_local** represents an offset starting from an undetermined global point. At the end of the second step, we determine this global point as **bkt_rebase_offset**. Therefore, we can finalize **bucket_offset** = **bucket_off_local** + **bkt_rebase_offset**.

In Section 5, we report metrics from our *two-stage-counter* PSH algorithms. Our *two-stage-counter* PSH algorithms further localize memory traffics within *GPU tiles* and boost the throughput.

10.2. ThunderKittens

THUNDERKITTENS [34] is a CUDA library that provides convenient tools to implement DL kernels that demand TensorCore throughputs, such as FlashAttention algorithms [6, 7, 32]. Original FlashAttention algorithms have highly complex implementations and require lots of engineering hours for proper implementations[4].

THUNDERKITTENS [34] simplifies FlashAttention implementations by providing numerical operations and matrix multiplications on 16×16 matrix tiles. Our implementation uses an early version of THUNDERKITTENS released on May 2024[5].

Current version of THUNDERKITTENS [34] incorporates a Load-Compute-Store-Finish (LCSF) pipeline to further optimize for newer GPUs including H100. LCSF rectifies the asynchronous producer-consumer paradigm of THUNDERKITTENS. With LCSF updates, THUNDERKITTENS can oversubscribe resources within SM, better saturate TensorCores, and outperform FlashAttention-3 in several settings [32, 34].

Unfortunately, Flash3D builds on top of FlashAttention-2 [6] and an early version of THUNDERKITTENS, and we did not have a chance to incorporate the latest version of THUNDERKITTENS. We aim to incorporate newer THUNDERKITTENS and FlashAttention-3 [32] in our future work to further boost Flash3D throughput and efficiency.

[4] https://github.com/Dao-AILab/flash-attention
[5] https://github.com/HazyResearch/ThunderKittens

10.3. Bucket-Swin

We describe the implementation details of our **Bucket-Swin** attention based on THUNDERKITTENS for Alg. 3 and Alg. 4. We focus on explaining the zero-overhead nature of **Bucket-Swin** attention. By "zero-overhead," we mean that our **Bucket-Swin** attention only incurs MHSA costs without additional memory or latency. Since THUNDERKITTENS loads inputs in tile-sized chunks, we achieve zero-overhead bucket-swin by simply loading into L1 the tiles corresponding to the set of buckets among which we wish to compute attentions, as detailed in the following paragraphs.

After our PSH algorithms, point features are represented in a contiguous array $\texttt{FP16}[N, d]$, where N is the number of points and d is the number of feature dimensions. Our point feature array is a concatenation of buckets along the N dimension. Therefore, any bucket-size-aligned sub-array represents a bucket. For example, when bucket size is 512, a sub-array $\texttt{FP16}[512 : 1024, d]$ contains all feature vectors of a bucket.

The goal of **Bucket-Swin** attention is to associate arbitrary buckets to an attention scope and compute MHSA within this logical scope. Notably, our **Bucket-Swin** attention does **not** introduce additional shuffling of the feature array. For a point i, its input features are located at $\texttt{FP16}[i, d]$ and its output features are located at $\texttt{FP16}[i, d]$ as well. Such fixed layout is the key to our zero-overhead **Bucket-Swin** attention.[6]

We describe how to vary attention scopes without shuffling point feature indices. As described above, a bucket-aligned sub-array represents a bucket. Consider an example of computing MHSA among two buckets: B_i and B_j, where B_i, B_j include contiguous point indices of buckets i, j respectively. Before MHSA, we map point features $\texttt{FP16}[i, d]$ into $Q \in \mathbb{R}^{N \times d}$, $K \in \mathbb{R}^{N \times d}$, and $V \in \mathbb{R}^{N \times d}$. Point features of B_i are $\texttt{FP16}[B_i, d]$. Query-key-value triplets of B_i are $Q[B_i, d]$, $K[B_i, d]$, and $V[B_i, d]$. We enforce that bucket sizes are all multiples of 16 so $Q[B_i, d]$, $K[B_i, d]$, and $V[B_i, d]$ can be tiled into 16×16 submatrices by THUNDERKITTENS.

FlashAttention-2 [6] operates on 16×16 tiles. As long as memory layouts for Q, K, V arrays suit 16×16 tiling, FlashAttention-2 operates on the original principles and assumptions. In our example, Q, K, V arrays for buckets B_i and B_j support 16×16 tiling. Therefore, we have full knowledge of tile addresses for buckets B_i and B_j. To compute MHSA output for B_i under a scope of $\{B_i, B_j\}$, we fetch tiles from $Q[B_i, d]$, $K[B_i, d]$, $K[B_j, d]$, $V[B_i, d]$,

[6] In Figure 3, we illustrate bucket-swin by pointing memory blocks to the destination memory blocks to best convey the conceptual model. In practice, we don't incur physical memory block rewriting but instead implement address redirection during FlashAttention-2 computation. Therefore, we have fixed memory layout and zero-overhead bucket-swin attention.

Algorithm 3 Bucket-Swin Attention Forward Pass

Require: BF16 matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, bucket size B, scope indices, and other parameters.

Divide $\mathbf{Q}$ into buckets according to scope indices and bucket size (instead of fixed B_r blocks); similarly divide $\mathbf{K}, \mathbf{V}$ using the bucket scheme.
Divide the output $\mathbf{O} \in \mathbb{R}^{N \times d}$ and logsumexp $L \in \mathbb{R}^N$ into corresponding buckets.
for each bucket index i **do**
  Load $\mathbf{Q}_i$ from HBM to on-chip SRAM.
  Scale $\mathbf{Q}_i$ by $\frac{1}{\sqrt{D}}$.
  On-chip, initialize $\mathbf{O}_i^{(0)} \leftarrow \mathbf{0}$, $\ell_i^{(0)} \leftarrow \mathbf{0}$, $m_i^{(0)} \leftarrow -\infty$.
  for each corresponding bucket index j **do**
    Asynchronously load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM using tile load.
    Compute $\mathbf{S}_i^{(j)} = \mathbf{Q}_i \mathbf{K}_j^T$.
    Compute $m_i^{(j)} = \max\left(m_i^{(j-1)}, \text{rowmax}(\mathbf{S}_i^{(j)})\right)$ and $\tilde{\mathbf{P}}_i^{(j)} = \exp\left(\mathbf{S}_i^{(j)} - m_i^{(j)}\right)$.
    Compute $\ell_i^{(j)} = e^{m_i^{(j-1)} - m_i^{(j)}} \ell_i^{(j-1)} + \text{row\_sum}(\tilde{\mathbf{P}}_i^{(j)})$.
    Update $\mathbf{O}_i^{(j)} \leftarrow \left(e^{m_i^{(j-1)} - m_i^{(j)}} \odot \mathbf{O}_i^{(j-1)}\right) + \tilde{\mathbf{P}}_i^{(j)} \mathbf{V}_j$, using MMA-based row scaling and matrix multiplication.
  end for
  Compute $\mathbf{O}_i = \text{diag}\left(\ell_i^{(T_c)}\right)^{-1} \mathbf{O}_i^{(T_c)}$.
  Compute $L_i = m_i^{(T_c)} + \log\left(\ell_i^{(T_c)}\right)$.
  Write $\mathbf{O}_i$ and L_i to HBM.
end for
return $\mathbf{O}$ and L.

Bucket Indexing Formulas: Let $q_{\text{buck\_seq}} = \lfloor qo\_blk\_iter / (\text{blocks\_per\_bucket}) \rfloor$, $q_{\text{buck\_id}} = \text{scope\_inds}[q_{\text{buck\_seq}}]$, and $q_{\text{buck\_blk\_off}} = qo\_blk\_iter \bmod (\text{blocks\_per\_bucket})$.

$V[B_j, d]$ to L1 cache blocks. The fetching step does not incur overheads other than plain FlashAttention-2 [6]. The only difference to FlashAttention-2 is redirecting tile addresses according to our bucket scheme. Then the computation and output steps are the same as FlashAttention-2. Therefore, our **Bucket-Swin** attention incurs no overheads on top of FlashAttention-2.

10.4. Fused FlashAttention-2 and Training

We precisely define our fused zero-overhead **Bucket-Swin** attention in this section. We describe both forward and backward passes for **Bucket-Swin** attention. The backward pass takes no special treatment since **Bucket-Swin** attention acts like common windowed attentions.

We provide the **Bucket-Swin** attention forward pass in Alg. 3 and the backward pass in Alg. 4. We omit the `cp.async` pipelining procedures from Alg. 3 and Alg. 4 for best clarity.

10.5. In-bucket Pooling

In Section 4.2, we motivate to replace the grid pooling operation [40] by our in-bucket pooling to capitalize on the existing geometric locality of buckets. Our in-bucket pooling provides higher throughput by removing global serialization and neighborhood finding. In addition, our in-bucket pooling has a fixed pooling ratio to offer smoother memory access patterns that can be achieved on GPU tiles. In this section, we describe further algorithm and implementation details of our in-bucket pooling.

Our main PSH algorithm establishes a linear order based on globally visible counters. However, our in-bucket pooling sub-bucket construction operates within local SMs (1024 points for a ThreadBlock). In contrast to our main PSH rebalancing algorithm, our in-bucket pooling sub-bucket construction includes three steps: **initial sub-buckets**, **new sub-bucket allocation**, **find new sub-bucket**.

Initial sub-buckets Similar to our main PSH algorithm, we use a hash function to assign each point a **subbuck_id**. We keep a ThreadBlock counter to allocate **subbuck_id**. Since points have uneven distributions among sub-buckets, sub-buckets have points no less than the desirable capacity ρ. Similarly, allocated **subbuck_id** is no more than the desirable total sub-buckets.

New sub-bucket allocation Then we examine each sub-bucket to relocate points beyond the requested capacity ρ. For each point that's beyond the sub-bucket capacity ρ, we allocate a new sub-bucket with id **subbuck_id**. We treat all

Algorithm 4 Bucket-Swin Attention Backward Pass

Require: BF16 matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V}, \mathbf{O}, \mathbf{dO} \in \mathbb{R}^{N \times d}$ and vector $L \in \mathbb{R}^N$ in HBM; output gradients $\mathbf{dQ}, \mathbf{dK}, \mathbf{dV} \in \mathbb{R}^{N \times d}$; bucket size B, scope indices, and other parameters.

Divide $\mathbf{Q}, \mathbf{K}, \mathbf{V}, \mathbf{O}, \mathbf{dO}$, and L into buckets according to scope indices and bucket size (instead of fixed block sizes).
Initialize $\mathbf{dQ} \leftarrow \mathbf{0}$ in HBM and partition $\mathbf{dK}, \mathbf{dV}$ into corresponding buckets.
Compute $D = \mathrm{rowsum}(\mathbf{dO} \circ \mathbf{O}) \in \mathbb{R}^d$ and partition D accordingly.
for each bucket index j over K/V blocks **do**
    Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
    Initialize on-chip accumulators: $\mathbf{dK}_j \leftarrow \mathbf{0}, \mathbf{dV}_j \leftarrow \mathbf{0}$.
    for each corresponding bucket index i over Q blocks **do**
        Load $\mathbf{Q}_i, \mathbf{O}_i, \mathbf{dO}_i, \mathbf{dQ}_i, L_i$, and D_i from HBM to on-chip SRAM.
        Compute bucket indices for Q as $q_{\text{buck\_seq}} = \lfloor q\_blk\_iter / (\text{blocks\_per\_bucket}) \rfloor$, $q_{\text{buck\_id}} = \text{scope\_inds}[q_{\text{buck\_seq}}]$, and $q_{\text{buck\_blk\_off}} = q\_blk\_iter \bmod (\text{blocks\_per\_bucket})$.
        Compute tile offsets based on these bucket indices and the warp ID; set pointers for the current tiles of $\mathbf{Q}, \mathbf{O}, \mathbf{dO}, \mathbf{dQ}$.
        Asynchronously load these tiles using tile load (with double-buffering as needed).
        On-chip, compute $\mathbf{S}_i^{(j)} = \mathbf{Q}_i \mathbf{K}_j^T$.
        Compute $\mathbf{P}_i^{(j)} = \exp(\mathbf{S}_i^{(j)} - L_i)$, where the subtraction uses the bucket's logsumexp value.
        Update $\mathbf{dV}_j \leftarrow \mathbf{dV}_j + (\mathbf{P}_i^{(j)})^T \mathbf{dO}_i$.
        Compute an intermediate gradient $\mathbf{dP}_i^{(j)} = \mathbf{dO}_i \, \mathbf{V}_j^T$.
        Compute $\mathbf{dS}_i^{(j)} = \mathbf{P}_i^{(j)} \circ (\mathbf{dP}_i^{(j)} - D_i)$.
        Update $\mathbf{dQ}_i \leftarrow \mathbf{dQ}_i + \mathbf{dS}_i^{(j)} \, \mathbf{K}_j$.
        Update $\mathbf{dK}_j \leftarrow \mathbf{dK}_j + (\mathbf{dS}_i^{(j)})^T \, \mathbf{Q}_i$.
        Write the updated $\mathbf{dQ}_i$ back to HBM.
    end for
    Write the accumulated $\mathbf{dK}_j, \mathbf{dV}_j$ back to HBM.
end for

Ensure: $\mathbf{dQ}, \mathbf{dK}, \mathbf{dV}$.

points until **subbuck_id** reaches the total sub-bucket number. At this point, sub-buckets allocated in the first step should contain exactly ρ points. And each newly allocated sub-bucket should contain 1 point.

Find new sub-bucket In this step, we treat the remaining points in this ThreadBlock. For each point, we scan all newly allocated sub-buckets and find one that has points closest to its coordinate. Then we try to commit this point into the newly found sub-bucket by `AtomicInc`. However, a sub-bucket might have reached its capacity before this point joins. It manifests in the result returned by `AtomicInc` being greater than or equal to ρ. Then we synchronize all threads to identify filled sub-buckets and under-filled sub-buckets. Finally, we repeat this step until all points find a sub-bucket.

Note that we repeat for multiple times of linear scans of newly allocated sub-bucket, which might seem to be a significant cost. However, all of our in-bucket pooling algorithms are confined within SMs running on fast local L1 cache blocks. Operations on local L1 cache blocks are significantly faster than those interacting with globally visible memory. Hence, our sub-bucket construction has higher throughput than our main PSH algorithm.

Based on our sub-bucket construction, we reduce point features within each sub-bucket to complete our in-bucket pooling operations as describe in main sections.

11. Community Adoption Guidelines

We recognize that integrating innovative methods like Flash3D into established communities, products, and pipelines involves both risk and cost. To assist in this transition, we offer our experiences and recommendations as gentle, practical guidelines.

11.1. Sub-Manifold Convolution Distillation

For users interested in distilling features from existing sub-manifold convolution pipelines [4] into Flash3D Point Transformers, we recommend incorporating fine-grained supervision from intermediate layers, not merely relying on final task objectives. Although a significant architectural gap exists between sub-manifold convolutions and Flash3D,

an important observation is that Flash3D consistently associates each token feature with a corresponding 3D point at every layer. In contrast, sub-manifold convolutions typically assign a feature vector to each voxel. This natural one-to-one correspondence between point tokens and voxel boxes provides a straightforward path for feature distillation from networks like MinkUNet [4] into Flash3D.

11.2. Early Sensor Fusion

There is growing interest in early sensor fusion across modalities—combining LiDAR point clouds, color images, videos, Radar traces, and pseudo-point-clouds. Our PSH algorithms are designed to adapt to the content and spatial distribution of points, ensuring that each bucket is filled with spatially proximate tokens. This property enables Flash3D attention scopes to seamlessly incorporate multi-modal tokens, even when their coordinates are imprecise, provided that they exhibit some degree of spatial proximity.

Flash3D can be configured in both pyramidal and U-Net architectures, making it versatile for multi-modal fusion across different tasks. Our empirical studies focus on the U-Net form for semantic segmentation and detection, while the pyramidal form is well-suited for classification and summarization tasks. As detailed in Section 4.1, each stage of Flash3D encapsulates a specific granularity and locality structure; hence, we recommend introducing heterogeneous tokens at the beginning of each stage, allowing PSH to automatically determine token affinity.

For outdoor scenes, we suggest integrating heterogeneous tokens at the third stage, where the point stride is typically 8 or 12 (depending on the second-stage reduction ratio). At this stage, each token represents 8 or 12 raw points, aligning with the token granularity found in other modalities, such as image and video patches [8]. This strategy not only streamlines sensor fusion but also ensures robust, universal feature representations across modalities.

11.3. Orin/Thor-U Deployment

We appreciate potential concerns in deploying Flash3D onto edge platforms and embedded solutions since our paper heavily focuses on server GPU investigations. However, we point out **the only chip requirement** for Flash3D is that the block-wise L1 cache size shall be greater than `100KB`. Orin GPUs have `192KB` block-wise L1 cache [16] and suffice to run Flash3D as is. Thor-U and Thor-X GPUs have larger block-wise L1 cache than Orin does and suffice to run Flash3D as well.