Speed3R: Sparse Feed-forward 3D Reconstruction Models

Supplementary Material

1. Efficient Kernel Implementation

1.1. Implementation Details

Our Gated Sparse Attention (GSA) utilizes a dedicated kernel for each branch. For the compression branch, we adapt the official FlashAttention implementation in Triton by integrating a streaming top-k selection, which employs a bitonic search algorithm, directly within the online softmax computation. The pseudocode for this forward kernel is presented in Algorithm 1.

The selection kernel shares a similar idea with block-sparse attention of original Flash Attention2. However, it adds a custom API to retrieve top-k indices from the compression kernel and to mask the first-frame tokens for VGGT, as shown in Algorithm 2. We do not show the pseudocode of the backward kernel, but we implement it following original NSA [50] and FlashAttention2 [8] logic.

We observe that the two branches of our proposed method are conceptually analogous to two recent training-free approaches.

Our compression branch shares its underlying philosophy with the token merging strategy of FastVGGT [32]. Both methods aim to reduce sequence length by merging redundant tokens to improve efficiency. However, they differ in their implementation: FastVGGT employs a bipartite matching algorithm for merging, whereas our approach utilizes a more efficient token average pooling mechanism.

Similarly, our selection branch aligns with the strategy used in Block-Sparse attention [36], which also selects top-k key blocks for each query to reduce computational overhead. A limitation of their work is the reliance on a pre-existing kernel [51] that lacks an implementation for the backward pass. This makes their method non-differentiable and thus non-trainable, hindering its scalability and practical utility. In contrast, our method introduces a custom, fully differentiable kernel, enabling end-to-end training.

To support the broader research community, we will open-source our kernel implementation, offering a scalable and trainable solution for efficient attention.

Algorithm 1 Compression Branch Forward Pass

Require: Query Q, Key K, Value V
Require: Top-k value k_{top}, Block dimensions B_M, B_N, Head dimension d_h
Ensure: Output O, Top-k indices I_{topk}

procedure GSA_FORWARD_DETAILED(Q, K, V, k_{top})
  for each query block Q_i of size $B_M \times d_h$ **in parallel do**
    Initialize output accumulator $O_i \leftarrow \mathbf{0}$
    Initialize row-wise max statistic $m_i \leftarrow -\infty$
    Initialize row-wise log-sum-exp statistic $l_i \leftarrow \mathbf{0}$
    Initialize top-k scores and indices: $(\mathcal{S}_i, \mathcal{I}_i) \leftarrow (-\infty, \text{null})$
    for each key/value block (K_j, V_j) of size $B_N \times d_h$ **do**
      Compute attention scores $S_{ij} \leftarrow Q_i K_j^T$
      ▷ 1. Update attention output via online softmax
      $m_{ij} \leftarrow \text{rowmax}(S_{ij})$
      $P_{ij} \leftarrow \exp(S_{ij} - m_{ij})$
      $l_{ij} \leftarrow \text{rowsum}(P_{ij})$
      $m_i^{\text{new}} \leftarrow \max(m_i, m_{ij})$
      $l_i^{\text{new}} \leftarrow e^{m_i - m_i^{\text{new}}} l_i + e^{m_{ij} - m_i^{\text{new}}} l_{ij}$
      $O_i \leftarrow e^{m_i - m_i^{\text{new}}} O_i + e^{m_{ij} - m_i^{\text{new}}} (P_{ij} V_j)$
      $m_i \leftarrow m_i^{\text{new}}$; $l_i \leftarrow l_i^{\text{new}}$
      ▷ 2. Update top-k indices via streaming selection
      $(S_{ij}^{\text{top}}, I_{ij}^{\text{top}}) \leftarrow \text{TopK}(S_{ij}, k_{\text{top}})$
      $(\mathcal{S}_i, \mathcal{I}_i) \leftarrow \text{Sort}((\mathcal{S}_i, \mathcal{I}_i), (S_{ij}^{\text{top}}, I_{ij}^{\text{top}}), k_{\text{top}})$
    end for
    Normalize final block output $O_i \leftarrow O_i / l_i$
    Store block output O_i into O and final top-k indices $\mathcal{I}_i$ into I_{topk}.
  end for
end procedure

2. Speed3R-VGGT Benchmarking

We benchmark the inference acceleration ratio of Speed3R-VGGT, as shown in Table 8. Our method achieves a 10.9x speedup compared to full attention and demonstrates greater efficiency than training-free methods. The latency is measured in pose-only mode. Note that in all implementations, we utilize VGGT with redundant memory storage removed, as proposed in FastVGGT [32], to benchmark the latency of VGGT. This optimization achieves an acceleration ratio of approximately 2x compared to the vanilla VGGT implementation.

Table 8. **Inference Time for Different Models.** Mean latency (in seconds) for varying sequence lengths on VGGT backbone. Our method achieves a 10.9x speedup on sequences of 1024 images.

Seq Length	32	64	128	256	512	1024
Full Attn.(VGGT)	0.79	1.81	5.44	18.21	67.27	271.21
FastVGGT [32]	0.63	1.25	2.59	6.48	18.30	61.33
Block Sparse [36]	0.53	1.16	2.89	8.16	26.15	75.42
Ours	0.46	0.87	1.78	3.88	9.16	24.85

Algorithm 2 Selection Attention Forward Pass

Require: Query Q, Key K, Value V, Top-k indices tensor T
Require: Block dimensions B_M, B_N, Head dimension d_h, Top-k count k_{top}
Ensure: Output O, Log-Sum-Exp values L

procedure BLOCKSPARSE_FORWARD
  for each query block Q_i of size $B_M \times d_h$ **in parallel do**
    Initialize output accumulator $O_i \leftarrow \mathbf{0}$
    Initialize row-wise max statistic $m_i \leftarrow -\infty$
    Initialize row-wise sum-exp statistic $l_i \leftarrow \mathbf{0}$
    for k from 0 to $k_{\text{top}} - 1$ **do**
      Load key block index $j \leftarrow T_{i,k}$
      Load corresponding key block K_j and value block V_j
      Compute attention scores $S_{ij} \leftarrow Q_i K_j^T$
      ▷ Update attention output via online softmax
      $m_{ij} \leftarrow \text{rowmax}(S_{ij})$
      $P_{ij} \leftarrow \exp(S_{ij} - m_{ij})$
      $l_{ij} \leftarrow \text{rowsum}(P_{ij})$
      $m_i^{\text{new}} \leftarrow \max(m_i, m_{ij})$
      $l_i^{\text{new}} \leftarrow e^{m_i - m_i^{\text{new}}} l_i + e^{m_{ij} - m_i^{\text{new}}} l_{ij}$
      $O_i \leftarrow e^{m_i - m_i^{\text{new}}} O_i + e^{m_{ij} - m_i^{\text{new}}} (P_{ij} V_j)$
      $m_i \leftarrow m_i^{\text{new}}$; $l_i \leftarrow l_i^{\text{new}}$
    end for
    Compute final Log-Sum-Exp $L_i \leftarrow m_i + \log(l_i)$
    Normalize final block output $O_i \leftarrow O_i / l_i$
    Store block output O_i into O and statistics L_i into L.
  end for
end procedure

3. Training/Evaluation Details

Table 9. Summary of Datasets Used in Training and Testing

Dataset Name	Scene/Sequence Count	Downsample Ratio
Co3Dv2 [25]	10764	2x
Hypersim [26]	461	1x
Scannetpp [49]	996	2x
DL3DV [19]	6,378	4x
ARKitScenes [1]	3,344	2x
WildRGBD [45]	2,753	5x
vkitti [5]	5	1x

Datasets. The statistics of our training datasets are detailed in Table 9. To manage storage requirements, we downsampled each sequence at the ratios specified in the table. During training, we sample 2-24 images following previous works. For the sampling process, we employed random sampling for most datasets. However, for the VKITTI2 dataset, which is characterized by its long sequences, we adopted the sampling strategy from π^3 [43]. The total size of the processed training data amounts to approximately 2TB, excluding depth maps. Our distillation-based strategy obviates the need to store these large depth files, resulting in significant storage savings.

Speed3R-VGGT Training Loss. Due to the self-normalization nature of VGGT, we do not need to normalize prediction and teacher supervision. The Speed3R-VGGT is trained end-to-end with a composition of two terms: $\mathcal{L}_{\text{camera}}$, $\mathcal{L}_{\text{depth}}$. The camera loss, $\mathcal{L}_{\text{camera}}$, supervises a set of N camera pose predictions ($\hat{g}i$) by comparing them against their teacher model predictions (g_i) using the Huber loss. The depth loss, $\mathcal{L}$depth, implements an aleatoric uncertainty loss that leverages a predicted uncertainty map, Σ_i^D, which acts as a measure of confidence. This uncertainty map is used to weigh the discrepancy between the predicted depth ($\hat{D}i$) and pseudo depth GT from the teacher model (D_i), and it also weighs an additional gradient-based term ($\nabla \hat{D}i - \nabla D_i$) to preserve structural details. The depth loss formulation, $\mathcal{L}\text{depth} = |\Sigma_i^D \odot (\hat{D}_i - D_i)| + |\Sigma_i^D \odot (\nabla \hat{D}_i - \nabla D_i)| - \alpha \log \Sigma_i^D$, therefore penalizes depth and gradient errors based on the model's own confidence while also regularizing the uncertainty prediction itself.

Speed3R-π^3 Training Loss. Analogous to the training of Speed3R-VGGT, the loss function for Speed3R-π^3 is a composite objective comprising two primary components: one for the depth head and another for the pose head. A key aspect of this loss, adopted from the π^3 methodology [43], is the normalization of both the model's predictions and the teacher supervision signals by the average depth value computed across all views. The overall loss function adheres to the original π^3 implementation, incorporating several specific terms: an aligned depth loss, a surface normal loss derived from MoGe [41], and a relative pose loss that is computed on all pairs of predicted camera poses.

Train from Scratch. In section 5, we train our sparse model and dense model from scratch. We train both models with 8 GPUs for 40 epochs with gradient accumulation of 2. We initialize both models' encoders with DINOv2 [23] weights and the other parts with Kaiming Initialization [14].

4. Sparsity Calculation

For our method and Block-Sparse [36], we report the sparse ratio based on the selection stage, as both the query (q) and key (k) are downsampled by a factor of 16 in the compression stage. This results in a negligible additional FLOPs cost.

For FastVGGT [32], we report their threshold-based sparse ratio. Their method sparsifies both the query (q) and key (k), which helps reduce FLOPs during the attention calculation. However, it introduces an additional pre-filtering stage, where the maximum similarity score of each token is computed relative to the protected tokens. This step adds quadratic complexity to their method.

Additionally, FastVGGT sets the first view as protected tokens that cannot be evicted. As a result, when the sparse ratio is set to 0.9, the actual effective sparse ratio is approximately 0.81. From a FLOPs perspective, we report their sparse ratio while accounting for this behavior and the overhead introduced by the pre-filtering stage.