A.1. Table of Contents

This supplementary material includes:

A.2. Additional Related Works

In this section, we provide an overview of token reduction and KV Cache compression strategies related to improving the efficiency of vision and language models.

Token reduction. Techniques that reduce tokens in runtime have emerged as promising approaches to accelerate the throughput of transformer architectures, however often come at the expense of performance degradation [19, 44, 48], or costly retraining [19, 38, 57]. Many of the past works in this domain, have focused on reducing the computational complexity of vision transformer models through token pruning [12, 23], merging [3, 17], or learning new token representations [48]. Despite promising approaches to vision encoders, with the rise of VLM architectures, it is natural to question if these methods can retain their generalizable performance and latency improvements. Reducing tokens at the visual encoder stage can be quite successful in speeding up the entire VLM workflow. In particular, Token Merging [3] merges tokens in the ViT, however, this was shown to scale poorly in complex vision language tasks [60]. In VILA [24], the authors showed that 2x down-sampling can help significantly after the vision encoder, hence speeding up both context and generation stages. However, any further native reduction caused significant performance degradation. Subsequently, works such as LLaVA-PruMerge[39], HIRED [1] and VisionZip [55] were proposed to "selectively" reduce tokens after the visual encoder. The refined approach to token pruning taken by these methods reduced the performance gap. However, at higher sparsities, they suffer significant performance degradation due to the erosion of query-related context, as shown in Table 1. Ultimately, this led to the rise of query-aware approaches such as FastV [4] and SparseVLM [60]. By delaying pruning until the LLM stage, the methods gain the advantage of leveraging query information to minimize the performance gap. Unfortunately, this approach does not scale to the real-world setting of multi-turn conversation, as shown in Figure 1. Hence, there is a clear gap in performance that neither query-aware or query-agnostic can solve in their own right. Addressing these limitations, our proposed framework divides the total sparsity load between context stage pruning in a query-agnostic manner and a decoding stage query-aware retrieval system. By selecting a highly sparse set of visual tokens to participate in the decoding stage, our approach can yield significant end-to-end speedup while maintaining superior performance in multi-turn settings. Another orthogonal line of research explored using language models to compress visual tokens (VoCo-LLaMA [56]). In their work, VoCo-LLaMA introduced visual compression during the instruction tuning phase to condition the model for faster inference, meanwhile, SparseVILA employs a training-free approach to inference acceleration in multi-turn conversation.

KV Cache Compression. In the realm of long-context LLM inference and serving, the large size of the KV cache introduces substantial time and memory overhead, necessitating the need for effective KV compression techniques. Recent works have proposed various algorithms to manage and reduce KV cache size while maintaining model accuracy. StreamingLLM [51] addresses infinitely long text sequences by preserving attention sinks and maintaining a finite KV cache in the recent window. SnapKV [22] predicts important tokens within a specified observation window prior to generation, helping to minimize unnecessary storage. H2O [62] selects a limited budget of essential KV cache entries based on cumulative historical attention scores. These methods discard portions of the KV cache based on past attention scores or current context; however, tokens discarded in this process may still hold relevance for future tokens, potentially resulting in information loss and degraded performance in multi-turn conversation tasks. To address this limitation, Quest [42] estimates approximate attention scores by computing upper-bound attention scores for each KV cache page, selectively preserving important KV entries. LazyLLM [11] takes a deferred approach by selectively computing KV pairs only for tokens crucial to the next token prediction, postponing the computation of other tokens until they become relevant. DuoAttention [50] differentiates between retrieval and streaming attention heads, applying a complete KV cache to retrieval heads while maintaining a lightweight, fixed-length KV cache for streaming heads. These approaches effectively reduce latency during decoding or prefilling stages by focusing on critical tokens. However, these methods primarily focus on text-based applications and overlook potential optimizations for image and video understanding tasks. Such tasks often exhibit greater sparsity due to the inherent temporal and spatial structure of video data, which provides further opportunities for performance enhancement.

A.3. Query-Aware Oracle

In this section, we detail the implementation of the query-aware oracle used in Section 2.2. In particular, we deliberately chose to use an oracle as it should, in theory, represent the upper bound of *any* query-aware method that could be introduced. Hence, if the limitation of multi-turn scaling applies to an oracle, it is a safe assumption that query-aware approaches will exhibit the same challenges.

Design. We include the implementation of our oracle in Algorithm 1. The query-aware oracle uses a greedy approach to converge to the optimal, query-informed tokens that guide the LLM in producing consistent responses with the vanilla implementation. By iterating through the configurations greedily, this method is quite costly in time, however, it serves as an ideal upper bound for current and future query-aware approaches as it factors in the question and original LLM response in order to guide the selection of tokens.

Algorithm 1 Query Aware Oracle Design

```
sys, visual, query = input_tokens
# Obtain Initial Output
response = llm(cat((sys, visual, query), 1))
# Embed response
emb = embed(response)
# Define token indices:
num_v = visual.size(1)
token_indices = torch.arrange(num_v)
token_mask = torch.ones(num_v, dtype=bool)
while num_removed < sparsity * visual.size(1):
    scores = {}
    shuffled_indices = shuffled(token_indices)
    for token in shuffled_indices:
        if token_mask[token] = False:
            continue
        #* remove token and check response
        token_mask[token] = False
        c_vis = visual[:, token_mask]
        new_r = llm(cat((sys, c_vis, query), 1))

        # cosine embedding comparison
        token_score = cos_dist(emb, embed(new_r))
        token_mask[token] = True
        scores[token] = token_score

    worst_token = min_key(scores)
    #* remove token:
    token_mask[worst_token] = False
    num_removed += 1

#* Obtain pruned response:
visual = visual[:, token_mask]
response = llm(cat((sys, visual, query), 1))
```

A.4. SparseVILA: Additional details

In this section, we provide additional details regarding SparseVILA's implementation and design choices.

A.4.1. Context Stage Pruning

In this section, we provide an overall algorithm to prune visual tokens in the context stage. Algorithm 2 presents the default implementation of SparseVILA for context stage pruning with a SigLip [58] visual encoder.

Algorithm 2 Context Stage Pruning after the visual encoder

```
# Tensor Shapes: (B, H, S, S) | (B, H, S)
attn_score, hidden_states = input
# Compute Outer Sum
attn_score = attn_score.sum(dim=(0,1))
# Compute Mean-Column Sum
salience = attn_score.sum(dim=1) / S
# Obtain question agnostic token mask
idx = salience > salience.quantile(sparsity)
# Prune Unimportant Tokens
hidden_states = hidden_states[:, idx]
```

Considerations for CLIP [37]. As stated in Section 3.1, when a *cls* token is present, we simply adapt the computation of `Mean-Column Sum` as: $salience = attn\_score[0, 1 :]$, essentially determining the influence that all sequence tokens have on the class token.

Considerations for multi-scale/frame samples. Multi-Frames and Multi-scale data often increase the inherent batch size of the sample. In this case, it no longer makes sense to remove the same token indices across different frames or scales. Hence, we simply restrict a consistent number of tokens to be removed but not which indices are selected. The result prevents the formation of a jagged tensor as the same number of tokens are selected per sample along the batch. We feel that more advanced strategies could better aid this performance, i.e. considering temporal relationships, however, we leave this for future works.

A.4.2. Generation Stage Retrieval

In this section, we provide an overall algorithm to retrieve visual tokens in the generation stage. Algorithm 3 presents the default implementation of SparseVILA for decoding stage token retrieval, compatible with FlashAttention2 [6].

Algorithm 3 Single Layer Query Aware KV Selection

```
# Tensor Shapes: (B, H, Q_len, D) | (B, H, KV_len, D)
q_proj, k_proj = input
d_k = q_proj.size(-1) ** 0.5
# Compute attn weights (B, H, Q_len, KV_len):
attn_scores = q_proj @ k_proj.transpose(-2, -1)
attn_weights = F.softmax(attn_scores / d_k, dim=-1)
# Extract Visual tokens using "start" and "end"
attn_vis = attn_weights[:, :, v_start:v_end]
# Compute Mean Column Sum
salience = attn_vis.sum(dim=(1,2)) / (H*Q_len)
# Obtain query aware KV indices
idx = salience > salience.quantile(sparsity)
# Create compact KV Cache
packed_kv = (k_cache[:,:,idx], v_cache[:,:,idx])
```

A.4.3. Additional details on KV cache compression techniques

Quest [42] Quest is a query-aware method to accelerate decoding throughput by selecting an active subset of the KV cache per generation step. By selecting a new KV subset for each generated token, Quest reports speedup only at very high context lengths, such as beyond 32K with Llama2 [45], whereas SparseVILA achieves speedup at just 3K context length. Rather than once per generation step, SparseVILA compresses the KV cache once per question/conversation turn, avoiding the latency overhead seen in Quest. As shown in Table A1, Quest performs worse than SparseVILA overall and degrades further at larger page sizes. Additionally, combining our query-aware strategy with Quest's generation strategy yields slightly worse performance than our SparseVILA while being significantly slower, suggesting that generation-token aware sparsity is unnecessary for efficient VLM inference.

Table A1. Compares and contrasts SparseVILA with LLM query-aware decoding strategy, Quest [42]. In all cases, we leverage SparseVILA's context pruning framework.

CTX Framework	GEN Framework	Context	Decoding	Sparse Attn	VideoMME
Vanilla	Vanilla	64K	64K	64K	59.7
SparseVILA	SparseVILA	16K	1K	1K	60.3
SparseVILA	Quest [42] Pg=128	16K	16K	1K	57.9
SparseVILA	Quest [42] Pg=16	16K	16K	1K	58.8
SparseVILA	SparseVILA + Quest [42] Pg=16	16K	4K	1K	59.4

SnapKV [22] SnapKV leverages an observation window towards the end of the context prompt in order to determine which parts of the KV cache should be kept for generation. Drawing a parallel to VLMs, the observation window can be seen as the question being pre-filled, in order to make this approach query-aware. However, a key contribution to their work is memory savings via discarding the original KV cache after selection. This would make it ill-suited for multi-turn conversation as they lose access to the original visual tokens. Additionally, they introduce larger overhead into the selection process with pooling operations. One advantage of their method is selecting tokens in a "head-aware" approach. Fortunately SparseVILA can also add this degree of freedom without incurring additional latency, by simply sorting two-dimensional IDs rather than averaging over the head dimension. Empirically we find that this added degree of freedom does improve performance on image-centric evaluation and is a viable extension of SparseVILA–SparseVILA + SnapKV [22]).

Table A2. Compares and contrasts SparseVILA with LLM observation-based decoding strategy, SnapKV [22]. In all cases, we leverage SparseVILA's context pruning framework.

CTX Framework	GEN Framework	Context	Decoding	DocVQA
Vanilla	Vanilla	3K	3K	74.4
SparseVILA	SparseVILA	1K	500	60.8
SparseVILA	SnapKV [22]	1K	500	59.3
SparseVILA	SparseVILA + SnapKV [22]	1K	500	61.5

A.4.4. Discussion on SoTA Overheads

By default, our latency comparisons in the main table do not consider the different overheads introduced by the complexity of each method, and instead simply consider the effect of reducing token on context and generation speeds. For Query-Aware methods, we do account that their pruning approaches are deferred until the third layer however we assume the cost of each strategy metric computation is negligible. In this section, we provide a deeper comparison of the CUDA-Time cost in (ms) that each method incurs in computing their metrics. The purpose of this analysis is to examine the real-world scalability of these methods beyond the latency gains from theoretical token reduction. Interestingly we find, that although methods such as SparseVLM [60] do achieve speedup from reducing the number of tokens, their overhead from metric computation and token recycling is quite large – hence they can only achieve speedup at a high enough token sparsity ratio. Likewise, methods such as PruMerge [39] incur significantly higher costs in their context stage pruning strategy due to the clustering of tokens in comparison with SparseVILA. Table A3 illustrates the low overhead cost associated with both context and decoding stage components of SparseVILA.

Table A3. Compares the context overhead of SparseVILA with other SoTA approaches on LLaVA-NeXT-7B [28] assuming a context length of 3K visual tokens and 100 query (question) tokens. Note, for FastV [4] and SparseVLM [60], we have already accounted for the first three dense layers in the main table speedup results, hence this additional overhead is not considered below. The default vanilla implementation incurs 425.8 ms context prefilling time and 1.89 s decoding time – overhead of the metrics are computed relative to this baseline.

Method	Module	Stage	Sparsity	CUDA Time (ms)	Overhead
PruMerge [39]	Vision Tower	Context	83.33%	132.8	31.2%
VisionZip [55]	Vision Tower	Context	83.33%	1.41	0.33%
HIRED [1]	Vision Tower	Context	83.33%	0.58	0.14%
SparseVILA	Vision Tower	Context	66.67%	0.87	0.20%
FastV [4]	LLM	Context	83.33%	21.1	5.0%
SparseVLM [60]	LLM	Context	83.33%	35.5	8.3%
SparseVILA	LLM	Decoding	90%	14.3	0.75%

A.4.5. Discussions on Sink & Retrieval Tokens

In this section, we discuss the emergence of sink and retrieval tokens throughout the VLM pipeline. In Figure A1 we explore the emergence of sink and retrieval tokens in the LLM. Profiling the attention maps in the LLaVA-1.5 architecture, we find that sink tokens and retrieval tokens are distributed over different layers of the model. In the early layers, we find there to be a strong focus on select tokens, which remain consistent regardless of the query. We additionally allude to this in Section A.8. As we progress deeper into the network, we begin to notice retrieval tokens emerge – i.e. query-dependent tokens. Nonetheless, the sink tokens remain, however at a lower strength. This insight influenced the design of SparseVILA by acknowledging that capturing both sink and retrieval tokens would require a full context stage (i.e. the computation of attention scores at every layer). Using this, we were able to design SparseVILA as a query-aware approach to select both sink and retrieval tokens simultaneously after the pre-filling/context stage. Previous methods that would prune tokens in early LLM layers, would be unable to capture retrieval tokens, as the attention patterns don't emerge until deeper in the network. Along this line, we compute the IoU of token indices selected over 38 questions in multi-turn on the GQA dataset. Interestingly, we find that using the scores from `Layer 2` retains roughly the same tokens over different queries, indicating a high sink token selection. If we shift this to `Layer 19` we are able to capture more retrieval tokens. Finally, SparseVILA runs the selection at every layer hence achieving a better balance between capturing sink and retrieval tokens.

A.5. Experimental Details

A.5.1. Benchmark Details

Following our description in Section 4, we provide further context on the datasets and benchmarks used. Our image benchmarks include tasks such as text recognition (TextVQA [40], DocVQA-Validation [35]), structuralized content understanding (InfoVQA-Validation [36], ChartQA [34]), general capability (MME [10]), mathematics (ScienceQA-Img [31]), hallucinations (POPE [21]), and complex visual question answering (GQA [15]). We evaluate SparseVILA on two long-context multiple-choice and visual question-answering video benchmarks (Video-MME [9] and MLVU [63]), in addition to a free-text long-generation `GPT-aided` evaluation on VideoChatGPT [33]. Finally, we perform analysis on the Visual Needle in a Haystack benchmark (V-NIAH) [53, 59]. For this benchmark, we used the 5 queries associated with `lmms-eval/needle` embedded in a long 2-hour+ video from previous works [53].

A.5.2. Baselines

Existing SoTA methods were primarily designed for single-round evaluation hence we had to adapt SparseVLM [60] and FastV [4] to multi-turn conversation. To support multi-round evaluation, we prefill the system, image, and initial query tokens to guide the visual token reduction for FastV [4] and SparseVLM [60]. The system and remaining image tokens are stored within the layer's KV cache to be re-used in each query associated with the target image.

A.5.3. Multi-Turn Evaluation Benchmarks

Algorithm 4 Multi-Round Evaluation Benchmark

```
# Input Tokens
system, visual, contexts = sample_tokens
# Prefill the model
kv_cache = model.prefill((system, visual))
# Number of system + image tokens
num_sys_img = kv_cache.size()
response = []
for q_num in range(len(contexts)):
    # Prefill Question q_num
    kv_cache = model.prefill(context[q_num], kv_cache)
    # Generate Response:
    kv_cache, resp = model.generate(kv_cache)
    response.extend(resp)
    # Evict Q&A from kv_cache
    kv_cache.reset_to(num_sys_img)
return response
```

As discussed in Section 4, we introduce a new evaluation framework to support the assessment of multi-turn conversation efficiently. As previously noted, our cache eviction strategy, under this setting, preserves question independence, promotes reliance on visual cues, and optimally increases the efficiency of the inference pipeline by removing redundant pre-filling operations. We include an algorithmic overview in Algorithm 4.

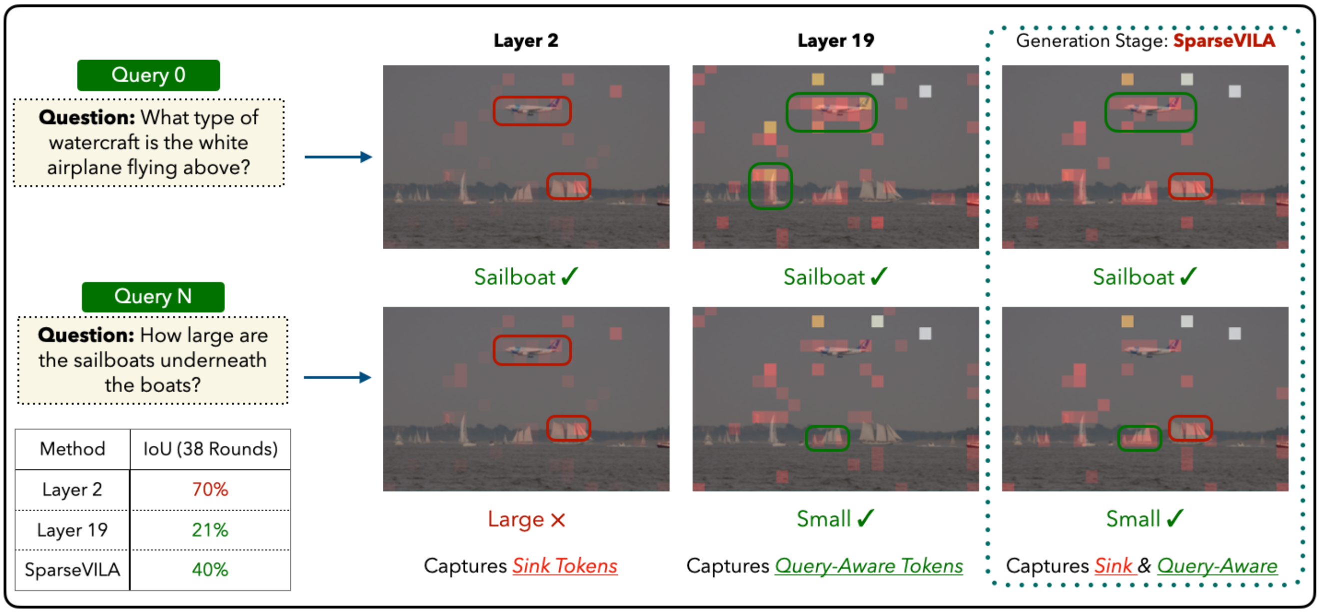

Figure A1. Visualization of the attention maps projected on the input image for LLaVA-1.5 [27] over different queries. By migrating sparsity into the decoding stage, SparseVILA enjoys the benefits of capturing different layer's attention scores, hence leveraging both sink and retrieval tokens.

VideoChatGPT Consistency

Prompt 1: <video>\n What game are the teams playing on the basketball court?

Vanilla: "The teams are playing dodgeball on the basketball court."

SparseVILA: "The teams are playing dodgeball on the basketball court."

Prompt 2: Can you describe the sports activity happening on the basketball court?

Vanilla: "The video shows a group of people playing dodgeball on a basketball court. They are wearing various colored clothes and are engaged in a game of dodgeball. The ball is thrown and caught by the players, and they continue to play."

SparseVILA: "The video shows a group of people playing dodgeball on a basketball court. They are wearing various colored clothes and are engaged in a game of dodgeball. The ball is thrown and caught by the players, and they continue to play."

Figure A2. Sample Qualitative Visualization of the Vanilla implementation and SparseVILA on the Consistency subset of the VideoChat-GPT [33] dataset for video captioning. This scene depicts a dodge-ball game being played on a basketball court. We have included two temporally sequential scenes above.

A.6. Quantitative Results

In this section, we include additional quantitative experiments validating SparseVILA's robustness on a variety of VLM architectures.

A.6.1. Image Benchmarks

In this section, we evaluate SparseVILA with query-agnostic approaches to further bolster our claims on compatibility and performance. We additionally include extensions of SparseVILA to frontier models.

A.6.1.1. Query Agnostic Comparisons

We evaluate SparseVILA against query-agnostic approaches of efficient VLM inference. Empirically we derive the following 2 findings: (1) At a given latency, SparseVILA will achieve better results due to the migration of sparsity from the context to the decoding stage, and (2) Applying SparseVILA to existing query-agnostic strategies can significantly improve their performance on image-centric benchmarks.

A.6.1.2. Frontier Models

We evaluate SparseVILA on several state of the VLM architectures including, LLaVA-1.5 [27], VILA-1.5 [24], Qwen2-VL [46], and the most recently released NVILA [30] architecture.

A.6.2. Video Benchmarks

We include shorter context length experiments with LongVILA [53] to support additional SoTA comparisons

Table A4. Extending SparseVILA with orthogonal context-stage pruning methods. Previous methods such as VisionZip [55] and HIRED [1], introduce carefully designed frameworks to leverage optimal token information from the vision tower. In this table, we compare the added benefit that SparseVILA brings to these strategies on image-centric benchmarks. We compare performance at a 1.48× speedup in E2E latency. In all cases, we outperform their implementations by introducing our decoupled sparsity framework to migrate reduction into the decoding stage.

	% of Visual Tokens		ChartQA	DocVQA^V	InfoVQA^V	Avg.
	Pre-filling	Decoding				Improvement (↑)
Max. Rounds #			5	13	35	
LLaVA-NeXT-7B [20]	100	100	50.7	74.4	37.7	–
+SparseVILA	33.33	10	39.9	54.4	30.8	–
+HIRED [1]	16.67	100	29.7	41.6	28.2	–
+SparseVILA	33.33	10	34.3	49.7	29.5	↑ 4.6%
+VisionZip [55]	16.67	100	34.1	51.2	30.6	–
+SparseVILA	33.33	10	36.6	53.8	30.4	↑ 1.7%

such as FastV [4] and SparseVLM [60]. We also include results on the NVILA [30] model in Table A6.

VideoChatGPT Temporal

Prompt 1: <video>\n What happens after the man cranks the lever on the sailboat?

Vanilla: "After the man cranks the lever on the sailboat, the camera turns to a white sailboat with a red sail sailing on the ocean."

SparseVILA: "After the man cranks the lever on the sailboat, the camera turns to a white sailboat sailing on the ocean with a red sail."

Figure A3. Sample Qualitative Visualization of the Vanilla implementation and SparseVILA on the Temporal subset of the VideoChatGPT [33] dataset for video captioning. In this particular sequence, the video depicts a man cranking a lever on the sailboat, followed by a visual of the boat sailing on the ocean. We have included two temporally sequential scenes above.

A.7. Multi-Turn V-NIAH

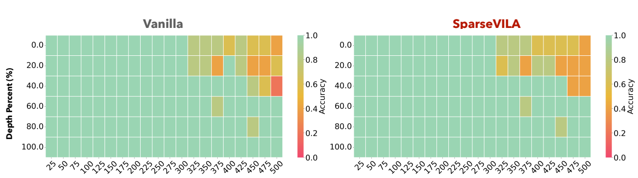

Figure A4. Comparing the baseline and SparseVILA on V-NIAH up to 500 Frames (128K context) on LongVILA-Qwen2 [53].

In this section, we provide additional details on how a multi-turn setting was constructed for the V-NIAH benchmark. We leverage the V-NIAH benchmark dataset from LongVILA [53] and LongVA [59]. There consists of 5 needles and N haystacks, where N is the number of frames. To facilitate a multi-turn setting, we embed all needles alongside the haystack. When prompting the LLM with one of the questions, the other needles act synonymously as haystack frames and should not affect the retrieval ability of the LLM if it can retrieve the correct frame. With query-aware pruning methods that remove tokens after the first query, the remaining needles may be pruned alongside the haystack, hence the model would exhibit performance degradation.

To account for the depth sensitivity, we run re-prefilling at each depth providing the first question, to re-prune the context and then prompt the target question to test the retrieval. In methods such as SparseVLM [60] or FastV [4], pruning in the context stages in a query-aware manner risks discarding the needle and hence being unable to respond to subsequent queries. In SparseVILA, we leverage heavy decoding sparsity which is query-aware and does not impact the KV cache that gets transferred over each turn of conversation, hence we can prune effectively per query without the risk of impacting future rounds. In Figure A4 we compare the long context performance of SparseVILA, and show retained performance up to 500 Frames on LongVILA.

A.8. Qualitative Visuals

In this section, we include additional qualitative visualizations of SparseVILA on the VideoChatGPT dataset. This benchmark features context lengths up to 64K and moderate generation lengths as a video captioning dataset. In several examples, we can see a positive correlation between the response produced by the Vanilla implementation and SparseVILA. In Figure A3 we show that SparseVILA preserves the primary message, despite a slightly different formatted output. Both responses resemble the same action and accurately depict the action portrayed in said video. In some cases, such as Figure A2, SparseVILA is able to exactly match the output of the Vanilla implementation over multiple rounds. Both the vanilla and SparseVILA are evaluated in a multi-turn setting for both prompts in Figure A2. Since both responses are correct, we intuitively illustrate VLM's inherent support for multi-turn conversation, and once again, demonstrate that SparseVILA can accurately and effectively support this setting, unlike previous query-aware pruning methods.

A.9. Visualizing Token Maps

In Figure A5 we include visualizations of the tokens pruned during the context stage, and those retrieved per query. As detailed in Section 4.5, the heatmap illustrates the frequency that a specific token was selected from the KV

Table A5. Image Benchmark. We extend SparseVILA to various Frontier Models on a multi-turn evaluation of 8 visual language benchmarks following the setting described in Section 4. † Qwen2VL-7B results are computed with `lmms-eval` within the range of [1, 12] million pixels.

		% of Visual Tokens	GQA	MME^{Sum}	SQA^{I}	POPE	VQA^{T}	ChartQA	DocVQA^{V}	$\text{InfoVQA}^{\text{V}}$	Avg.
	Max. Rounds #		92	4	11	18	2	5	13	35	
Llama-2	LLaVA-1.5-7B [24]	100	61.9	1866.73	69.6	85.4	57.9	20.8	27.6	25.7	55.3
	+SparseVILA	8.3	59.2	1758.1	68.5	84.8	57.1	18.8	24.5	24.8	53.2
	VILA-1.5-13B [24]	100	64.5	1843.1	80.0	85.8	65.2	52.1	58.3	36.1	66.8
	+SparseVILA	20	64.1	1850.2	79.8	85.5	65.1	48.7	54.2	33.8	65.5
Qwen2	Qwen2VL-7B [46]†	100	63.7	2065.4	–	88.6	78.0	72.9	OOM	OOM	81.3
	+SparseVILA	8.3	63.1	2036.6	–	87.4	77.0	68.9	OOM	OOM	79.6
	NVILA-8B [30]	100	64.9	2060	96.9	88.6	80.0	82.7	85.1	59.9	82.6
	+SparseVILA	25	64.8	2058	97.2	88.7	79.2	82.0	83.7	58.8	82.2

Table A6. Multi-Round Video Benchmarks. We present additional multi-round evaluations on 2 video benchmarks following the multi-turn setting described in Section 4. We show impressive retainment of performance on long-context models such as LongVILA [53].

	% of Visual Tokens		# Frames	VideoMME *w/o subtitle*				MLVU
	Pre-filling	Decoding		Overall	Short	Medium	Long	
	Max. Rounds #			3	3	3	3	20
NVILA-8B-Video [30]	100	100	256	63.7	75.0	62.3	53.9	70.3
+SparseVILA	75	25	256	62.0	74.9	59.6	51.6	69.4
+SparseVILA	50	25	256	61.3	74.3	59.1	50.3	69.0
LongVILA-Llama3-8B [53]	100	100	256	48.7	60.7	48.1	37.3	–
+SparseVILA	75	25	256	48.8	60.9	48.1	37.4	–
LongVILA-Qwen2-7B [53]	100	100	32	59.2	72.8	56.6	48.3	72.3
+FastV† [4]	50	100	32	58.9	72.0	56.6	48.2	70.4
+SparseVLM† [60]	50	100	32	58.1	70.8	54.9	48.7	69.3
+PDrop† [52]	50	100	32	59.0	70.8	57.1	49.2	–
+VisionZip [55]	50	100	32	59.1	71.3	56.4	49.4	69.3
+HIRED [1]	50	100	32	58.7	70.4	55.9	49.9	70.7
+SparseVILA	100	12.5	32	59.4	72.7	56.7	48.8	72.1
+SparseVILA	50	25	32	59.8	73.0	57.2	49.1	72.9

cache over all layers – hence we are visualizing the frequency selection of tokens in the image. Notably, SparseVILA is able to capture both *visual attention sinks* and *retrieval tokens* to support accurate and efficient generation. In the first case of Figure A5 we can see a strong retrieval for the man on the bike for the first prompt and the car/white sign on the right for the second prompt. Likewise, the second sample shows a crisp localization of the car in the first prompt.

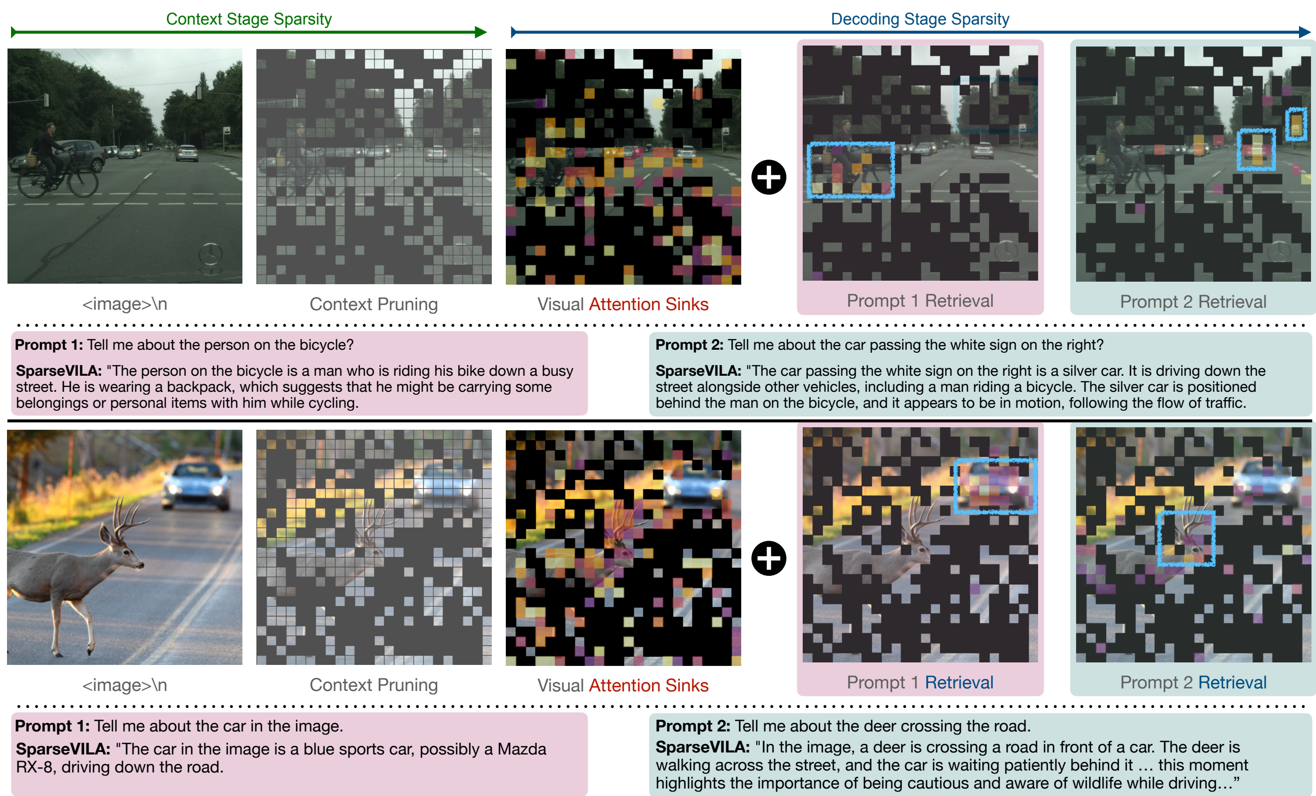

Figure A5. Visualization of selected tokens from the KV cache for various queries in the multi-round setting. The heatmap illustrates the frequency of token selection from the KV Cache for the particular query of interest. We conduct said visualization on LLaVA-1.5 [27]