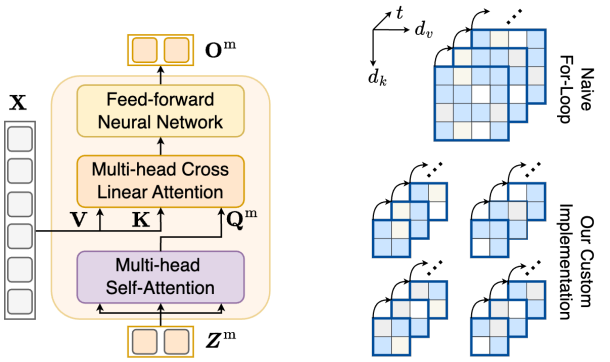
Scalable Video Action Anticipation with Cross Linear Attentive Memory

Supplementary Material



(a) CLAM architecture. (b) Our CUDA implementation.

Figure A.7. **Details of the proposed CLAM.** (a) Our CLAM diverges from the traditional transformer decoder by integrating a cross linear attention layer, enabling constant memory usage during inference. (b) Our implementation splits input tensors into smaller blocks. Each block processes the entire sequence independently, enabling parallel computation across blocks while retaining intermediate results in fast shared memory (SRAM).

This supplementary material provides further information for the main paper. Section A offers additional methodological and implementation details, Section B includes further ablations, and Section C discusses the limitations of the proposed model. The code is available on GitHub[1].

A. Additional Details

CLAM Implementation. The proposed CLAM (Fig. A.7a) retains the conventional transformer decoder architecture with a multi-head self-attention layer and a feed-forward network, but replaces standard cross-attention layer with our reformulated cross linear attention layer. We describe the forward and backward passes of our efficient CUDA implementation of the cross linear attention state update (Eq. (6)) in Alg. 1 and 2, assuming a single attention head for simplicity. As the state update involves only element-wise operations, we divide tensors into smaller blocks along the key (D_K) and value (D_V) feature dimensions, creating $\frac{D_K}{D_k} \cdot \frac{D_V}{D_v}$ blocks (see Fig. A.7b). In the forward pass, tensors are loaded block-by-block from HBM (high bandwidth memory) to SRAM (shared memory) for faster computation. The algorithm performs the sequential recurrence in parallel across all blocks. By retaining intermediate states within fast on-chip memory and minimizing redundant global memory access, our implementation maximizes parallelism across blocks while reducing latency. The backward pass utilizes the same efficient memory access pattern as the forward pass to compute gradients.

[1] https://github.com/zeyun-zhong/scalable-anticipation

Algorithm 1 Forward Cross Linear Attention State Update

Input: $\mathbf{G}, \mathbf{KV} \in \mathbb{R}^{T \times D_K \times D_V}$, $D_k \in [D_K]$, $D_v \in [D_V]$

$i_k,\ i_v \leftarrow$ Thread's unique index along D_K, D_V
$\mathbf{P} \in \mathbb{R}^{D_k \times D_v} \leftarrow$ Local block address based on i_k, i_v
$\mathbf{S}' \leftarrow \mathbf{0} \in \mathbb{R}^{D_k \times D_v}$ // Initialize block state
for $t \leftarrow 0$ **to** $T-1$ **do**
  $\mathbf{P}' \leftarrow \mathbf{P} + t \cdot D_K \cdot D_V$ // Global block address
  $\mathbf{G}' \leftarrow$ Load from HBM to SRAM based on $\mathbf{P}'$
  $\mathbf{KV}' \leftarrow$ Load from HBM to SRAM based on $\mathbf{P}'$
  $\mathbf{S}' \leftarrow \mathbf{G}' \odot \mathbf{S}' + \mathbf{KV}'$ // Compute on chip
  Store $\mathbf{S}'$ to HBM
end for

Streaming vs. Full-sequence Inference. In the main paper (Figs. 5 and 6), we evaluate our method in both *full-sequence* and *streaming* settings. These settings differ in how models process inputs: full-sequence approaches observe all frames jointly, whereas streaming approaches update predictions frame by frame. Fig. A.8 highlights the underlying architectural reason for the observed efficiency gap: quadratic-complexity models revisit all past tokens at each step, while our model propagates compact states with constant per-step complexity. This design explains why our method achieves nearly constant per-step runtime and memory usage in streaming scenarios, making it well-suited for real-time deployment.

Additional Implementation Details. Following prior works [1, 18, 80], we conduct our experiments on pre-extracted features. For EpicKitchens100, we resample videos at 30 FPS and then fine-tune TSN [64] on the classification task, as done in [18]. Additionally, we extract Swin-B RGB features for EK100 using an off-the-shelf checkpoint [24, 40, 80]. For Thumos14, videos are resampled at 24 FPS, with frames extracted at 4 FPS for training and validation. We use a two-stream TSN [64] pre-trained on Kinetics [5] to extract frame-level RGB and optical flow features. For Ego4D, we use VideoMAE [60] pre-trained on Kinetics [5] and finetuned on Ego4D by [6]. Using a sliding window approach, we feed 16 consecutive frames (0.533s of video at 30 FPS) into the model to generate one visual embedding. Since [6] fine-tuned separate models for verb and noun classification, we concatenate their embeddings and input the combined representation into our proposed model.

We employ the AdamW optimizer for training our

Algorithm 2 Backward Linear Attention State Update

Input: $\mathbf{dS}, \mathbf{G}, \mathbf{S} \in \mathbb{R}^{T \times D_K \times D_V}$, $D_k \in [D_K]$, $D_v \in [D_V]$

$i_k,\ i_v \leftarrow$ Thread's unique index along D_K, D_V
$\mathbf{P} \in \mathbb{R}^{D_k \times D_v} \leftarrow$ Local block address based on i_k, i_v
$\mathbf{dS}' \leftarrow \mathbf{0} \in \mathbb{R}^{D_k \times D_v}$ // Initialize block gradient
for $t \leftarrow T-1$ **to** 0 **step** -1 **do**
  $\mathbf{P}' \leftarrow \mathbf{P} + t \cdot D_K \cdot D_V$ // Global block address
  $\mathbf{P}'_{t-1} \leftarrow \mathbf{P}' - D_K \cdot D_V$ // Address at last timestep
  $\mathbf{dS}'_t \leftarrow$ Load from HBM to SRAM based on $\mathbf{P}'$
  $\mathbf{G}' \leftarrow$ Load from HBM to SRAM based on $\mathbf{P}'$
  $\mathbf{S}'_{t-1} \leftarrow$ Load from HBM to SRAM based on $\mathbf{P}'_{t-1}$
  $\mathbf{dS}' \leftarrow \mathbf{dS}' + \mathbf{dS}'_t$ // Update cumulative gradient
  $\mathbf{dKV}' \leftarrow \mathbf{dS}'$ // Update gradient for $\mathbf{KV}$
  $\mathbf{dG}' \leftarrow \mathbf{dS}' \odot \mathbf{S}'_{t-1}$ // Update gradient for $\mathbf{G}$
  $\mathbf{dS}' \leftarrow \mathbf{dS}' \odot \mathbf{G}'$ // Update gradient for $\mathbf{S}$
  Store $\mathbf{dG}'$ to HBM
  Store $\mathbf{dKV}'$ to HBM
end for

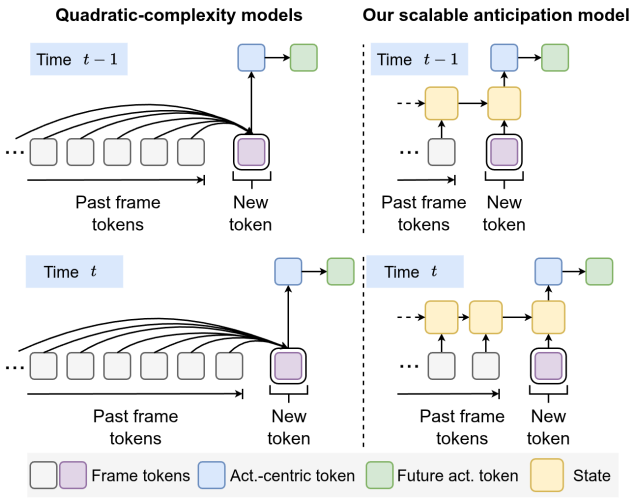


Figure A.8. Comparison of quadratic-complexity and our scalable anticipation model. Quadratic models grow in cost with context length, while our model propagates compact states, ensuring scalable inference with constant per-step complexity.

model. For EpicKitchens100, we train the model for 50 epochs with a learning rate of 5e-4, a batch size of 256, and observations set to 32 seconds. Ego4D follows a 15-epoch training schedule with a 1e-4 learning rate and a batch size of 128, with 32-second observation intervals. For Thumos14, the training spans 40 epochs with a learning rate of 1e-4, and the model processes entire videos as a single batch. For all datasets, the classifiers for observation and anticipation are shared.

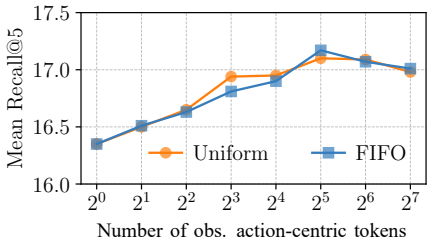


Figure B.9. **Impact of the number of observed action-centric tokens** on the validation set of EK100. No memory module is applied. Results tend to plateau after a certain number of tokens.

Temp. Agg.	EK100
GRU	15.64
LSTM	14.77
Mamba	**17.39** ± 0.11

(a) Temporal processor.

Act.& Mem. Emb.	Mean Recall@5
✗	17.09 ± 0.13
✓	**17.39** ± 0.11

(b) Embedding distinguishing action-centric and memory tokens.

# Layers	Mean Recall@5
1	15.89 ± 0.22
2	16.41 ± 0.22
4	**17.39** ± 0.11
6	17.13 ± 0.19

(c) Number of layers.

Hidden Dim	Mean Recall@5
256	15.33 ± 0.10
512	15.68 ± 0.06
1024	**17.39** ± 0.11
1280	16.94 ± 0.14

(d) Hidden dimension.

Table B.10. **Additional architectural analysis** on EK100. Metrics used are class-mean Top-5 Recall.

B. Additional Results

Are All Past Action-centric Tokens Necessary? Figure B.9 illustrates the impact of varying the number of action-centric tokens (act-tokens) on EK100, without applying the memory module. We either uniformly sample from observations (*Uniform*) or keep only the last tokens (*FIFO*). Performance plateaus or even slightly declines beyond a certain token count, likely due to high information redundancy, as each token is derived from all preceding frames, potentially adding no new useful information. Additionally, there is no significant difference in performance between the two sampling strategies. We use the FIFO design by default.

Temporal Processor Architecture. Table B.10a compares the performance of different temporal processors: GRU [11], LSTM [31], and Mamba [27]. On EpicKitchens100, Mamba significantly outperforms GRU and LSTM, demonstrating its effectiveness for complex, large-scale tasks.

Impact of Action-centric and Memory Embedding. Table B.10b evaluates the contribution of the action-centric and memory embeddings. Including these embeddings im-

Method	Linear Comp.	Frozen Enc.	Init.	Mean Recall@5		
				Action	Verb	Noun
AVT [23]	✗	ViT-B	IN21K	14.9	30.2	31.7
AFFT [80]	✗	Swin-B	K400	16.1	–	–
DCR [72]	✗	TSM	K400	16.1	32.6	32.7
MeMViT [67]	✗	MViTv2-24	K700	17.7	32.2	37.0
RAFTformer [22]	✗	MViTv2-24	K700	18.0	33.7	37.1
S-GEAR [15]	✗	ViT-B	IN21K	18.3	31.1	37.3
InAViT [52]	✗	Motionformer	IN21K	25.9	52.5	51.9
PlausiVL [43]	✗	ViT-G/14	–	**27.6**	**55.6**	**54.2**
Ours w/o Mem.	✓	Swin-B	K400	18.0	33.7	30.0
Ours	✓	Swin-B	K400	18.4	32.1	30.7

Table B.11. **Comparison to the state-of-the-art on EpicKitchens100 [14] validation** using different RGB encoders.

proves performance, highlighting their importance for the future predictor to distinguish between action-centric tokens and memory tokens.

Number of Temporal Processor Layers. Table B.10c analyzes the effect of the number of temporal processor layers on the performance. Increasing the number of layers improves performance up to 4 layers, achieving the highest class mean recall, after which additional layers do not improve performance.

Hidden Dimension. Table B.10d investigates the impact of the dimension of the hidden state on the performance. A larger dimension improves the performance, with a peak at 1024. The performance drops slightly when the dimension increases to 1280.

State-of-the-art Comparison on EK100. In Table B.11, we extend our evaluation to include methods utilizing advanced backbones. Equipped with a frozen Swin-B [24, 40] encoder, our model surpasses AFFT [80] by roughly 2% and achieves performance comparable to S-GEAR [15]. We also report results for InAViT [52] and PlausiVL [43] for completeness, although they operate under significantly higher computational budgets. InAViT relies on auxiliary hand-object detectors and region-specific attention, while PlausiVL leverages Large Language Models (LLMs) alongside a massive ViT-G/14 backbone. While these computationally intensive components yield higher recall, our framework prioritizes scalability. We achieve competitive performance among standard visual methods while maintaining linear computational complexity and avoiding the latency overhead of LLMs or external object detectors.

C. Limitations

While our Cross Linear Attentive Memory (CLAM) is applicable to a variety of video understanding tasks, its evaluation has primarily focused on the action anticipation task. Future research will explore its effectiveness across other video analysis applications. Additionally, although our model employs multiple memory queries to extract relevant context cues from past observations, the updated memory tokens do not inherently maintain a sequential order. This characteristic may potentially offer further advantages that have yet to be fully realized and optimized. Furthermore, while the non-autoregressive Transformer decoder used in our future predictor performs robustly across diverse datasets, it exhibits the highest latency compared to other components of our system (see Table C.12). Future improvements will aim to reduce this latency, thereby increasing the practical applicability of our model in real-time scenarios.

Component	Processor	Memory	Predictor
Runtime (ms)	150.55	105.36	300.19

Table C.12. **Cumulative runtime of each component** when inferring a video with 480 frames in a streaming fashion.