DET-SAM2: SELF-PROMPTING SEGMENTATION FRAMEWORK BASED ON SEGMENT ANYTHING MODEL 2

Zhiting Wang
wangzt@motern.com

Qiangong Zhou
zhouqg@motern.com

Zongyang Liu
lzy@motern.com

Shenzhen Motern Technology Co., Ltd.

ABSTRACT

Segment Anything Model 2 (SAM2) demonstrates exceptional performance in video segmentation and refinement of segmentation results. We anticipate that it can further evolve to achieve higher levels of automation for practical applications. Building upon SAM2, we conducted a series of practices that ultimately led to the development of a fully automated pipeline, termed Det-SAM2, in which object prompts are automatically generated by a detection model to facilitate inference and refinement by SAM2. This pipeline enables inference on infinitely long video streams with constant VRAM and RAM usage, all while preserving the same efficiency and accuracy as the original SAM2.

This technical report focuses on the construction of the overall Det-SAM2 framework and the subsequent engineering optimization applied to SAM2. We present a case demonstrating an application built on the Det-SAM2 framework: AI refereeing in a billiards scenario, derived from our business context. The project at `https://github.com/motern88/Det-SAM2`.

***K*eywords** Segment Anything Model 2 · Detect Model · Engineering Optimization

1 Introduction

Segment Anything Model 2 (SAM2)[1] is currently the state-of-the-art (SOTA) model in video segmentation. It exhibits advanced object-level instance segmentation capabilities and continues the mask fuzzy matching and interactive refinement features from Segment Anything (SAM)[2]. However, the official implementation of SAM2 necessitates user interaction with the initial frame of the video and the application of condition prompts before inference can commence. During the inference process, if corrections are required for any inaccuracies in SAM2's outputs, users must introduce new condition prompts adjacent to the erroneous frames and re-initiate the inference process. On one hand, frequent manual interactions constrains the applicability of SAM2 in automated scenarios. On the other hand, SAM2 requires a complete re-inference each time new condition prompts or categories are introduced, which significantly exacerbates the already substantial performance overhead associated with SAM2.

To address these challenges, we developed Det-SAM2. A video object segmentation pipeline that automatically generates prompts for SAM2 using the YOLOv8[3] detection model,This pipeline includes a post-processing step for SAM2's segmentation results, facilitating business decisions in specialized scenarios, all without human intervention. Det-SAM2 retains the robust object segmentation and refinement capabilities of SAM2 while eliminating the need for manual input of condition prompts, thereby enabling fully automated inference. Furthermore, we have implemented a series of engineering enhancements to the Det-SAM2 framework, which effectively reduce performance overhead.

Specifically, our key contributions include:

1. We have developed a self-prompting video instance segmentation pipeline (Det-SAM2-pipeline) that requires no manual interaction for prompt input. It supports inference and segmentation of specific categories, as determined by a custom detection model, using **video streams**. It returns segmentation results with accuracy comparable to that of SAM2, thereby facilitating post-processing for business applications.

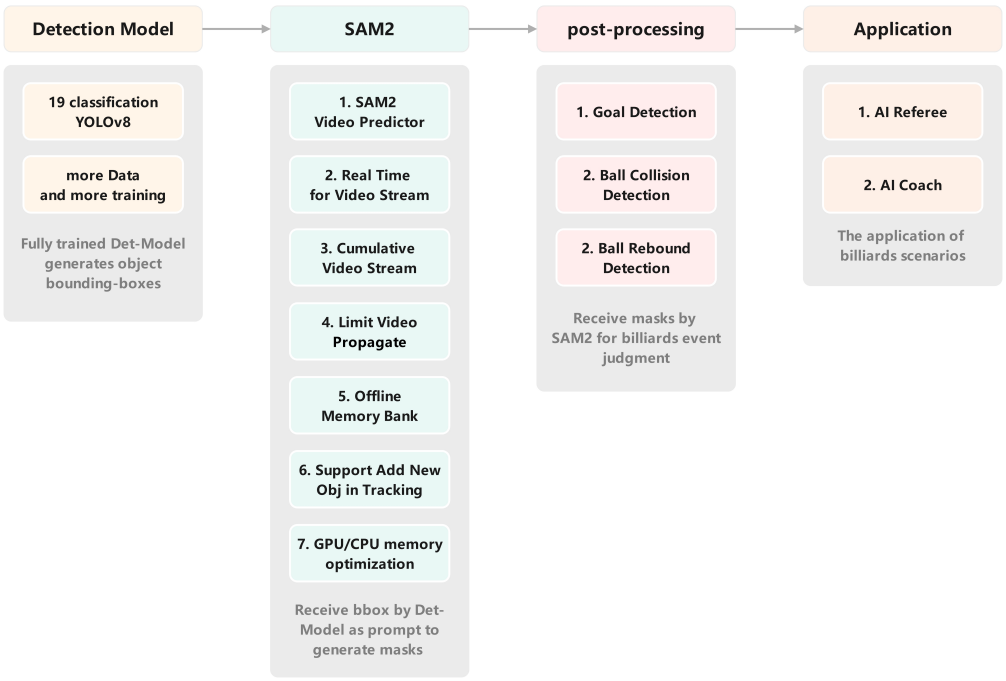

Figure 1: Overview of Det-SAM2 Tasks. The overall technical pipeline of Det-SAM2 comprises three key components: the detection module, the pixel-level video tracking module using SAM2 instances, and the post-processing module. The detection model provides initial (potentially imperfect) bounding boxes, which are used as conditional prompts for SAM2. The SAM2 video predictor propagates these discrete frame prompts (`propagate_in_video`) across all frames in the video, enabling continuous inference. Ultimately, the SAM2 video predictor outputs spatiotemporal masks for object instances throughout the video. The post-processing module then analyzes the obtained masks to deliver accurate and quantifiable results, thereby supporting higher-level applications such as an AI coach or AI referee in billiards scenarios.

2. We have incorporated the capability to **add new objects online** during the SAM2 inference tracking process without interrupting the inference state.
3. Our pipeline introduces the application of a memory bank from one video to a new video, referred to as a **preload memory bank**. This feature allows the system to leverage memories(object categories, shapes, motion states) from the inference analysis of the previous video to assist in conducting similar inference in a new video, **without the need to add any prompts** in the new video.
4. We have ensured **constant GPU and memory usage** in the Det-SAM2 pipeline, enabling the inference of infinitely long videos without interruption.

Our work focuses exclusively on engineering optimizations and does not involve training or fine-tuning the SAM2[1] model itself. The tasks associated with the implementation of the Det-SAM2 framework and the development of its application example(AI referee for the billiards scene) are illustrated in Figure 1.

2 Related Work

Segment Anything Model 2[1] achieves object segmentation in videos by leveraging its comprehensive understanding of object-level concepts. It effectively manages deformations such as stretching and occlusion, representing a significant advancement beyond traditional detection and segmentation models. Furthermore, SAM2 exhibits robust correction capabilities, allowing newly provided conditional prompts to be applied retroactively across all previously inferred frames, thereby facilitating the correction of specific errors. The SAM2 Video Predictor, as illustrated in Figure 2, maintains a Memory Bank that stores the input image features and predicted mask results for each frame. During the prediction of the segmentation mask for each frame, the inference computation incorporates the Memory Bank, conditional prompts, and input image features.

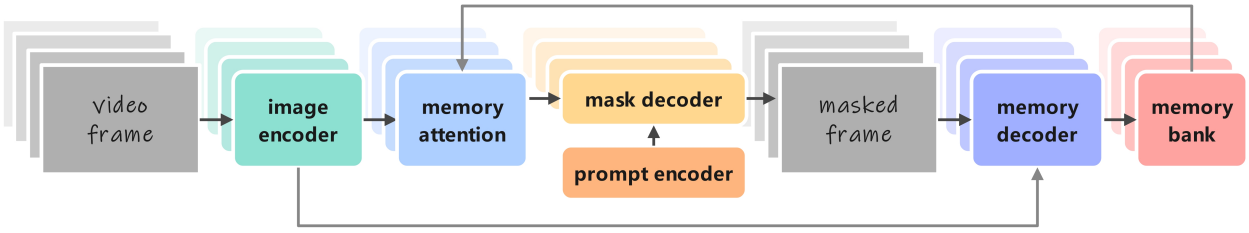

Figure 2: Original Framework of SAM2. The video frame features are processed through Memory Attention, integrating information from the current frame with that in the Memory Bank, and then passed to the Mask Decoder, which uses the conditional prompts to generate the predicted masks. The Memory Bank is extracted by the Memory Decoder from the conditional frames. The Memory Decoder receives outputs not only from the Mask Decoder but also from the Image Encoder.

YOLOv8[3] is a new version of the YOLO series launched by Ultralytics in 2023. It is built upon YOLOv5[4], incorporating significant architectural and methodological innovations. With its outstanding performance and broad applicability, YOLOv8 has emerged as the preferred choice for industrial deployment in the field of object detection.

3 Methodology

In this section, we focus on how we built the Det-SAM2 pipeline corresponding to the seven subtasks illustrated in Figure 1, as well as the series of engineering challenges encountered and addressed throughout this process.

3.1 Detection Model + SAM2 Video Predictior

As previously mentioned, interactive prompts are essential for SAM2's[1] accurate segmentation abut also pose a barrier to achieving fully automated inference without human intervention. The initial prompt for SAM2's first input frame must be manually provided to start the inference process. We explored the possibility of utilizing a detection model to eliminate the need for human-provided prompts.

As shown in Figure 3, compared to Figure 2, we introduced a branch that connects the video frame input to the prompt input. This branch is powered by a detection model, which applies prompts in the form of detection boxes to the video frames for our specified categories. With this addition, SAM2 can commence predictions without human intervention. This marks the initial form of Det-SAM2 (Detection Model + SAM2 Video Predictor).

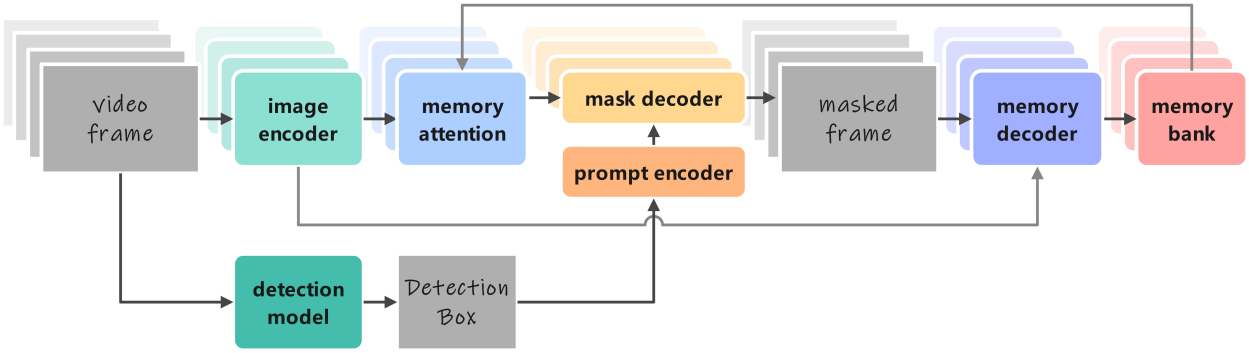

Figure 3: Det-SAM2 Experimental Demo Framework Diagram. The condition prompt for a given frame is automatically added, and the condition prompt is provided by the detection model (in this case, YOLOv8). The detection box results are used as the prompt input for the Prompt Encoder.

3.2 Det-SAM2 in Video Stream

Once our Det-SAM2 system is capable of initiating segmentation predictions without manual intervention, we also seek it to leverage SAM2's powerful correction capability. Rather than relying solely on the prompt information from the initial frame to infer the predictions for the entire video. We aim to continuously add box prompts automatically

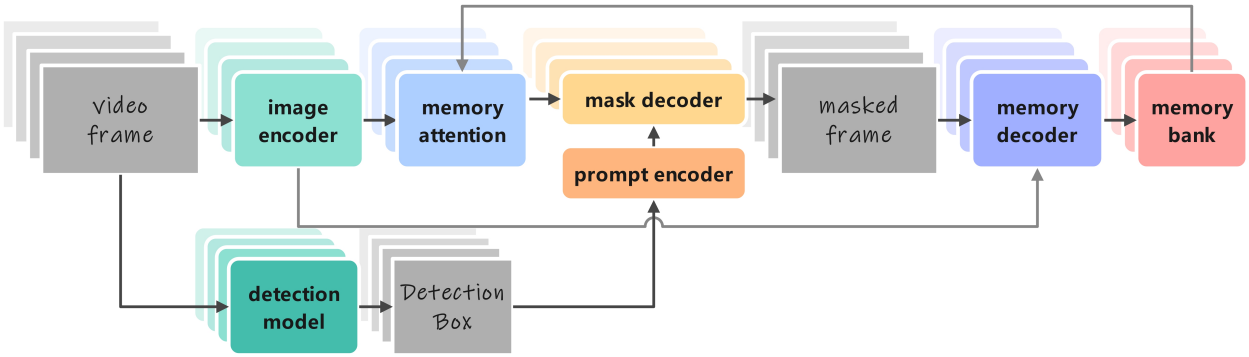

Figure 4: The Det-SAM2 framework facilitates for the automatic addition of condition prompts for each frame. In contrast to Figure 3, where the detection model branch is not only active in the initial frame of the video, it is now applied to every frame throughout the video.

generated by the detection model to SAM2 in the video stream. Due to SAM2's correction mechanism, whenever SAM2 receives a new prompt, it propagates the new prompt information to all previously inferred frames (using `propagate_in_video`). Consequently, the memory bank, now updated with the newly received prompt, is reintegrated into the memory attention of each previously processed frame, allowing for the recalculation of segmentation masks and enabling effective corrections.

The implement the automatic addition of condition prompts for each frame within the Det-SAM2 framework is illustrated in Figure 4. The primary distinction between this figure and Figure 3 is that we invoke the detection model branch for each frame. Additionally, the schematic representation of Det-SAM2 processing the video stream over time is depicted in Figure 5.

However, as shown in Figure 5, in practice, the propagation process (`propagate_in_video`) consumes a considerable amount of inference time. This is because, during the propagation, each time a new condition frame is added, SAM2 re-infers all previous frames. As a result, when processing a video of length N frames, the current framework requires inference for a total of $\frac{1}{2}N^2$ frames.

3.3 Cumulative Video Stream

A straightforward method to optimize inference overhead is to minimize the frequency of inference propagation (`propagate_in_video`). This reduction can be approached from two perspectives: One approach is to reduce the frequency at which condition frames are generated, as only these frames trigger SAM2's correction mechanism, leading to repeated inference propagation on previously processed frames. Another approach is to increase the number of video frames in a single input. When multiple frames are input simultaneously, SAM2 performs propagation only once.

To implement this, we aim to space out the inference of the detection model so that not every frame input into SAM2 requires to have condition prompts from the detection model. Additionally, We propose allowing new incoming video frames to accumulate in a video frame buffer while receiving the video stream. Instead of inputting each frame into the Det-SAM2 framework individually, we input a sequence of accumulated frames all at once. This way, SAM2 processes multiple frames in a single pass, thereby reducing the number of propagation steps.

The diagram illustrating the process of accumulating the video stream and spacing out the condition prompts from the detection model is shown in Figure 6.

By accumulating a certain number of video frames before inference, we can significantly reduce the number of propagation steps in SAM2. Suppose we accumulate a sequence of K frames at a time. When performing inference on an entire video of length N frames, the propagation process (`propagate_in_video`) would only need to process approximately $\frac{1}{2K}N^2$ frames.

3.4 Limited Video Propagate

We already know that SAM2[1] achieves prompt-based corrections by receiving new conditional prompts and performing re-inference propagation(`propagate_in_video`) across all historical frames. In the case of a static video (i.e., when the entire video is input at once), it is understandable that each propagation must be executed across all frames. However, during inference on a video stream, it is unnecessary for the propagation process to be applied to all historical frames.

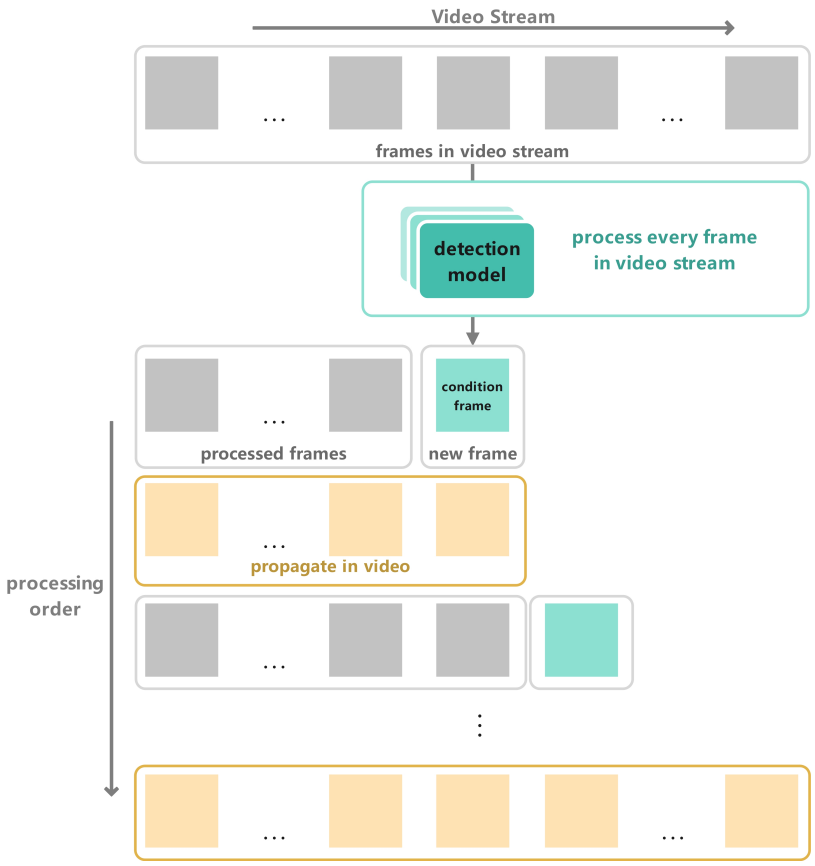

Figure 5: Det-SAM2 Video Stream Processing Diagram. Each frame passes through the Detection Model as a condition frame for SAM2 (represented in green in the diagram), and then the propagation operation (depicted in yellow as "propagate in video") is applied to all previously processed video frames to enable the correction capability.

In the actual Det-SAM2 process, the non-condition frames that require correction are typically not far removed from the condition frames they depend on. Therefore, we can limit the number of frames involved in the inference during each propagation. During each inference process in the video stream,most distant past frames are already finalized and do not require correction. In contrast, the more recent a past frame is, the more likely its inference results may be altered in future predictions. Therefore, we impose the following restrictions on the propagation(`propagate_in_video`):

1. Set the `propagate_in_video` to process frames in reverse order, starting from the most recent frame.

2. Limit the maximum number of frames to be processed during propagation(`propagate_in_video`). However, the length of the propagation should at least encompass the cumulative video frame sequence from the previous inference; otherwise, the propagation would lose its corrective significance.

The process diagram for Limited Video Propagation is illustrated in Figure 7.

Increasing the maximum propagation length (`max_frame_num_to_track`) can expand the correction range but will result in higher computational overhead. Conversely, reducing the maximum propagation length can accelerate inference speed but will limit the range within which conditional frames can correct non-conditional frames in the video stream.

By limiting the propagation length(`max_frame_num_to_track`) to M, and the cumulative frame buffer size (`max_frame_num_to_track`) to K, inference for a video of length N frames requires processing approximately $\frac{M}{K}N$ frames. This implies that when M and N are fixed, the actual number of frames processed by Det-SAM2 increases linearly with the video length.

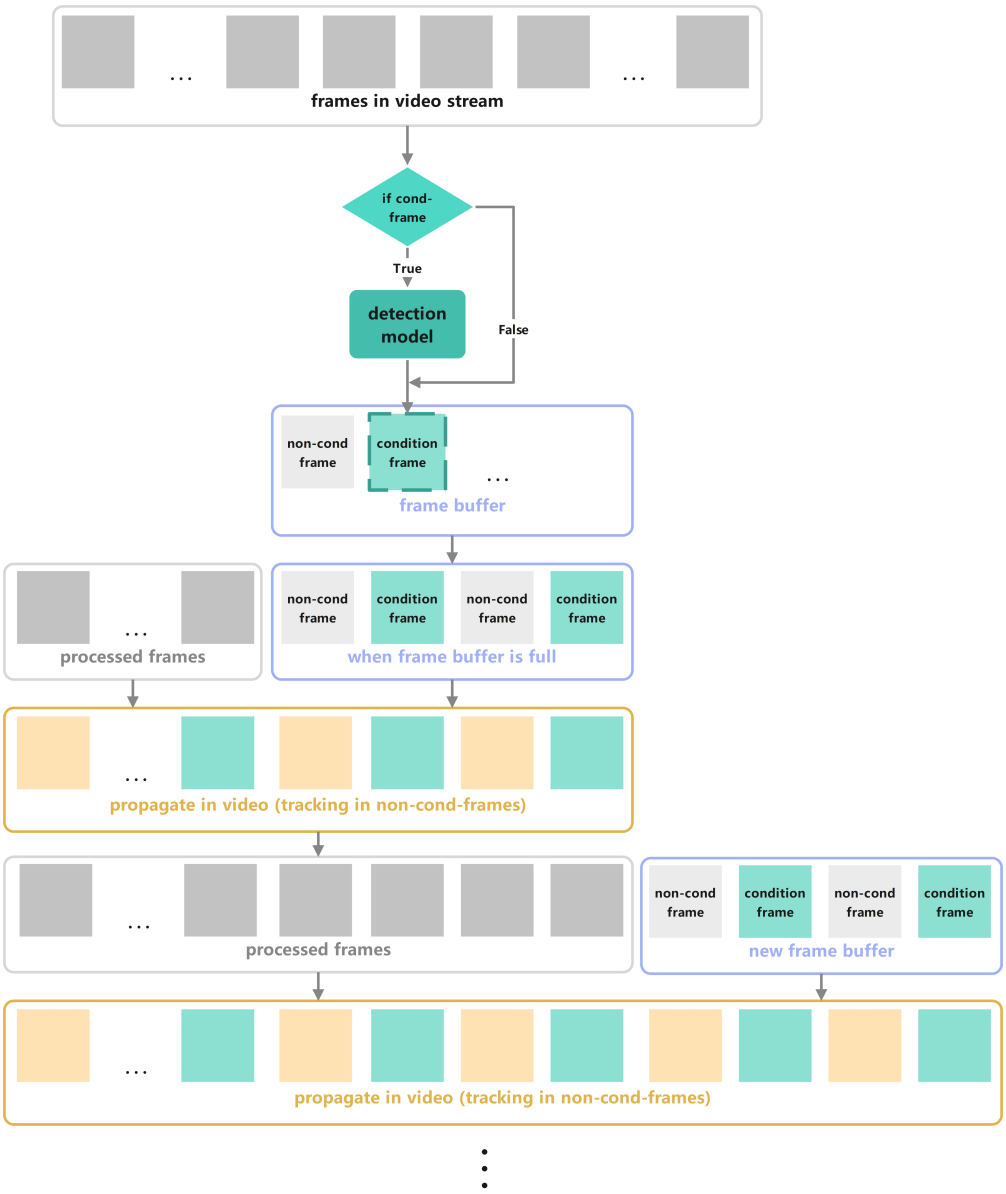

Figure 6: Flowchart of cumulative video stream and interval-based detection condition prompts in Det-SAM2. For each frame received in the video stream, it will first be accumulated in the frame buffer. When the frame buffer accumulates enough frames, we will input the sequence of frames from the buffer into the Det-SAM2 framework all at once. During inference of the current video frame sequence, SAM2 will determine which frames are condition frames and which are non-condition frames based on the interval setting. Condition frames are provided with condition prompts by the detection model, and the prompt embeddings are generated by SAM2's Prompt Encoder.

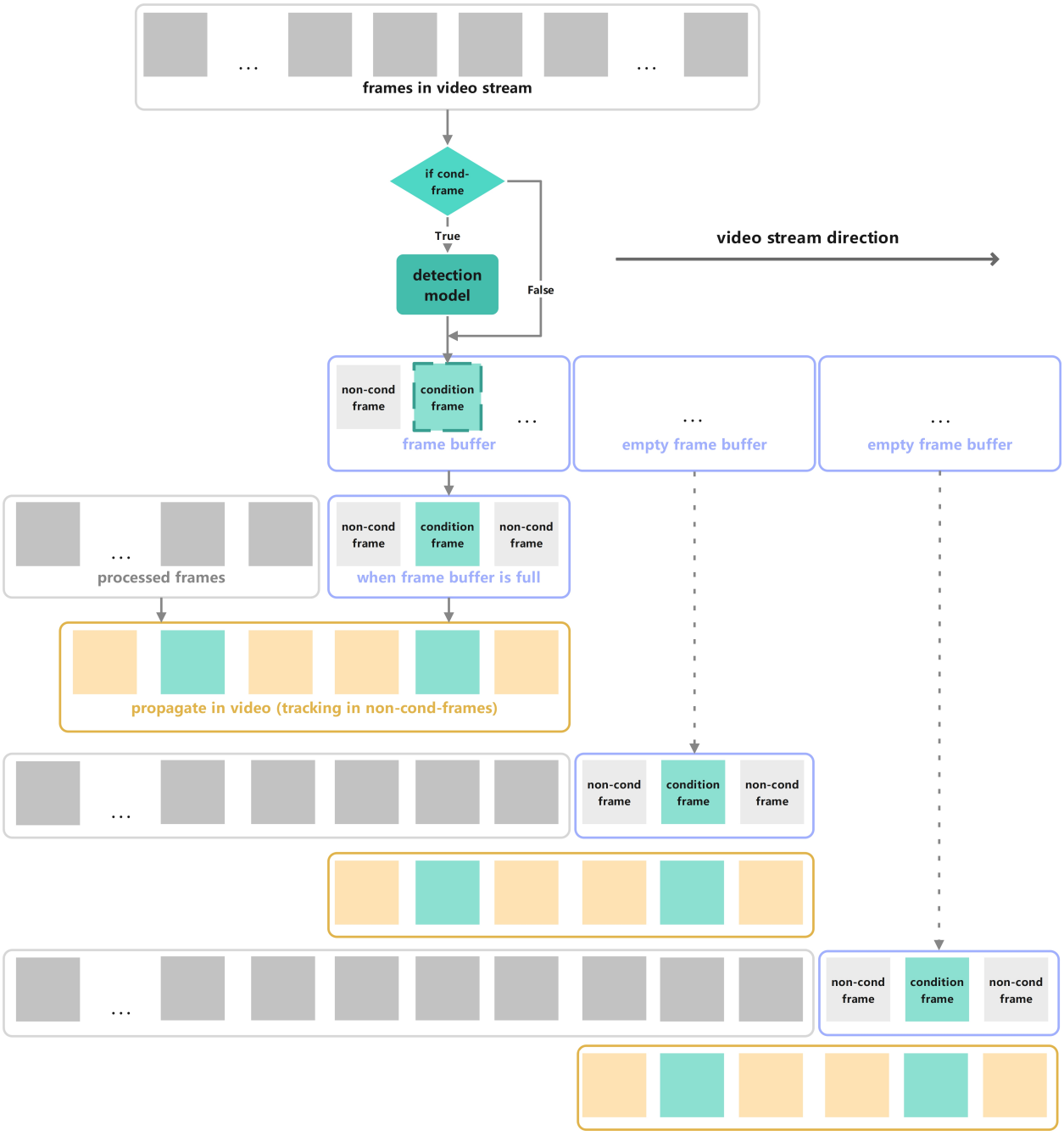

Figure 7: Flow Diagram of Limited Video Propagation: During propagation (propagate in video), the process is restricted to a user-defined maximum tracking length (`max_frame_num_to_track`), instead of iterating through all previous frames.

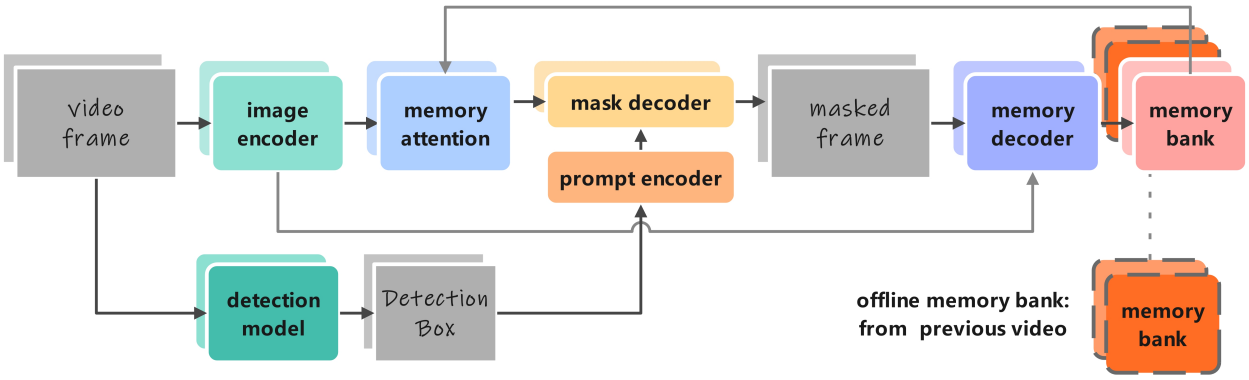

Figure 8: Preload memory bank flowchart. The Memory Bank preloads an offline memory bank, sourced from previously inferred videos. As a result, during the current inference, the previously inferred memory from past videos can be directly applied to the inference of the new video.

3.5 Preload Memory Bank(Offline Memory Bank)

In comparison to SAM[2], a significant enhancement in SAM2[1] that facilitates the transfer of image segmentation capabilities to video segmentation is the incorporation of the Memory Bank. This Memory Bank establishes associations between frames through Memory Attention, allowing SAM2 to leverage temporal context across video frames for improved segmentation performance. However, the generation and construction of the Memory Bank must be online, meaning it needs to be built in real-time during the inference process. Each time a new video frame is input, the Memory Bank is updated with the condition prompt, output mask, and video frame feature through the Memory Decoder.

Inspired by discussions in the official code repository issues[5], we propose preload a Memory Bank that has already been constructed from an old video. This approach enables the utilization of memory information from the previous video for the new video, allowing inference without the need to add any condition prompts. This entails allowing SAM2 to preload an offline Memory Bank that has been meticulously designed to include all necessary prompts and challenging sample prompts that may be required for the new video, similar to a "system prompt". During subsequent inference on the new video, the newly generated memory will be accumulated on top of the preloaded Memory Bank. Although the newly generated memory and the preloaded memory are conceptually distinct, there is no practical differentiation between the two during actual inference in SAM2. The process of preloading the memory bank is illustrated in Figure 8.

In the specific implementation, the `inference_state` in SAM2 holds all the information in the memory bank. Therefore, by migrating the `inference_state` to the new video inference, the new video can directly leverage the existing memory without needing to reinitialize the `init_state`.

3.6 Support Add New Objects in Tracking

Currently, in the official SAM2 implementation, it is only permissible to predefine the instances that need to be segmented before tracking begins, and it does not support the online addition of new instances during the tracking process. The specific reason is that when the memory bank is first initialized (`init_state`), the class mapping list is fixed. If new object categories that need to be inferred are introduced in a new frame, it results in a mismatch in the feature tensor sizes between the new frame and the old frames, making it impossible to compute.

n practice, during the real-time video stream processing with Det-SAM2, we often cannot predict which objects will appear in the future. Our detection model is likely to output only a partial set of categories when initializing the memory bank for the first time. Consequently, any remaining categories that are introduced to SAM2 after inference has commenced will be treated as new objects requiring processing. This situation is unavoidable. The official solution to this is to perform a `init_state`, which resets the memory bank. This means that in the frame where a new category appears, the memory bank is reinitialized. The consequence of this approach is that all previous inference results are lost, and segmentation inference starts anew from that point.

Moreover, performing a reinitialization each time a new category appears in a frame cannot guarantee that the reinitialized frame sequence will contain all potential categories. Therefore, any method that involves resetting the memory bank upon encountering a new category will inevitably trigger a `reset_state` initialization repeatedly in long videos. This continuous resetting to avoid tensor size mismatch errors between frames due to different numbers

of categories, and the subsequent loss of all previous inference results, effectively clears the memory bank each time, rendering the process inefficient and preventing the model from leveraging prior context across extended video sequences.

To address this issue, we propose enabling the natural addition of new object IDs during the tracking process without resetting the entire memory bank. We achieve this by updating the memory bank online during the tracking process. To enable the inference to naturally add new object categories during the tracking process, when encountering a new object category, we need to update the following operations:

1. Register a new ID's corresponding index list and information storage dictionary in the `inference_state`.
2. Update the memory bank of all previous frames using the new ID mapping table (reacquire `output_dict` or `temp_output_dict`, and generate the memory bank under the new ID mapping relationships through the Memory Encoder).

The schematic diagram of online updating of the memory bank during tracking is shown in Figure 9.

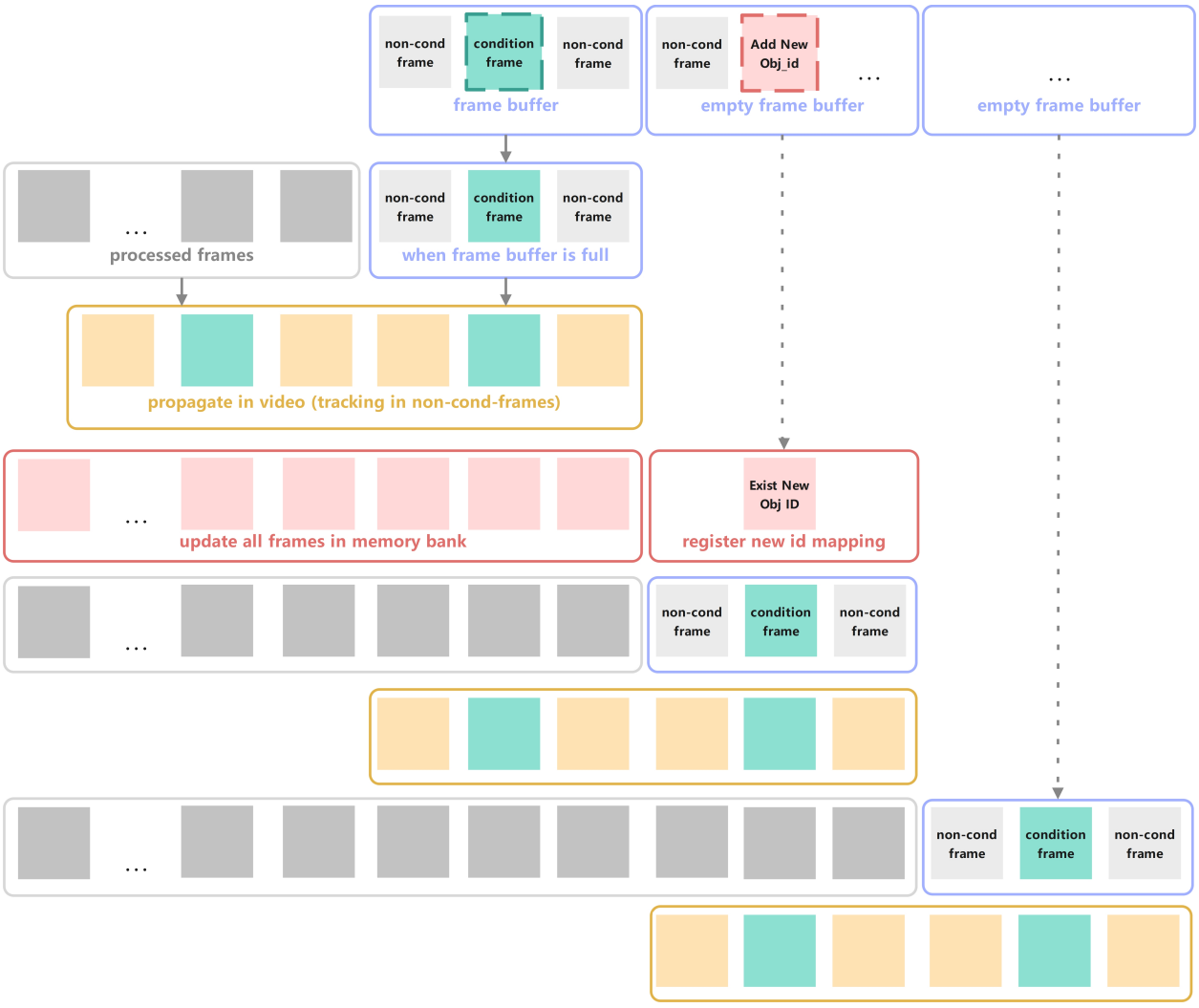

Figure 9: The schematic diagram of online memory bank updates when Det-SAM2 adds new objects during tracking. When the framework receives new object categories after initiating inference and tracking, it first registers the new object IDs in the memory bank. Subsequently, based on the updated ID mapping table, the memory bank for all previous historical frames is updated. (The batch dimension size of the information tensors in the memory bank for each frame depends on the number of IDs to be predicted. Without updating, tensor mismatches will occur, preventing the calculation of Memory Attention.)

After implementing the functionality shown in Figure 9, we now support the ability to add new object IDs online after tracking has started. However, in long video inference, the current method for online updating of the memory bank processes all historical frames in the memory bank using the memory encoder each time it is applied. If a new object appears at the end of a long video, this approach incurs a significant computational overhead.

To improve performance efficiency, we need to impose restrictions on two components:

1. Limit the number of frames updated in the memory bank when a new object category appears.

2. Limit the number of frames used for memory attention calculation to avoid using old frames that have not been updated.

The specific approach is as follows:

1. When a new category is added during inference and tracking, only a limited number of frames close to the current moment in the memory bank are updated, ensuring that all frames in the preload memory bank are also updated.

2. Restrict the number of frames used for memory attention calculation while ensuring that all condition frames in the preloaded memory bank are included in the computation.

The optimized process is shown in Figure 10.

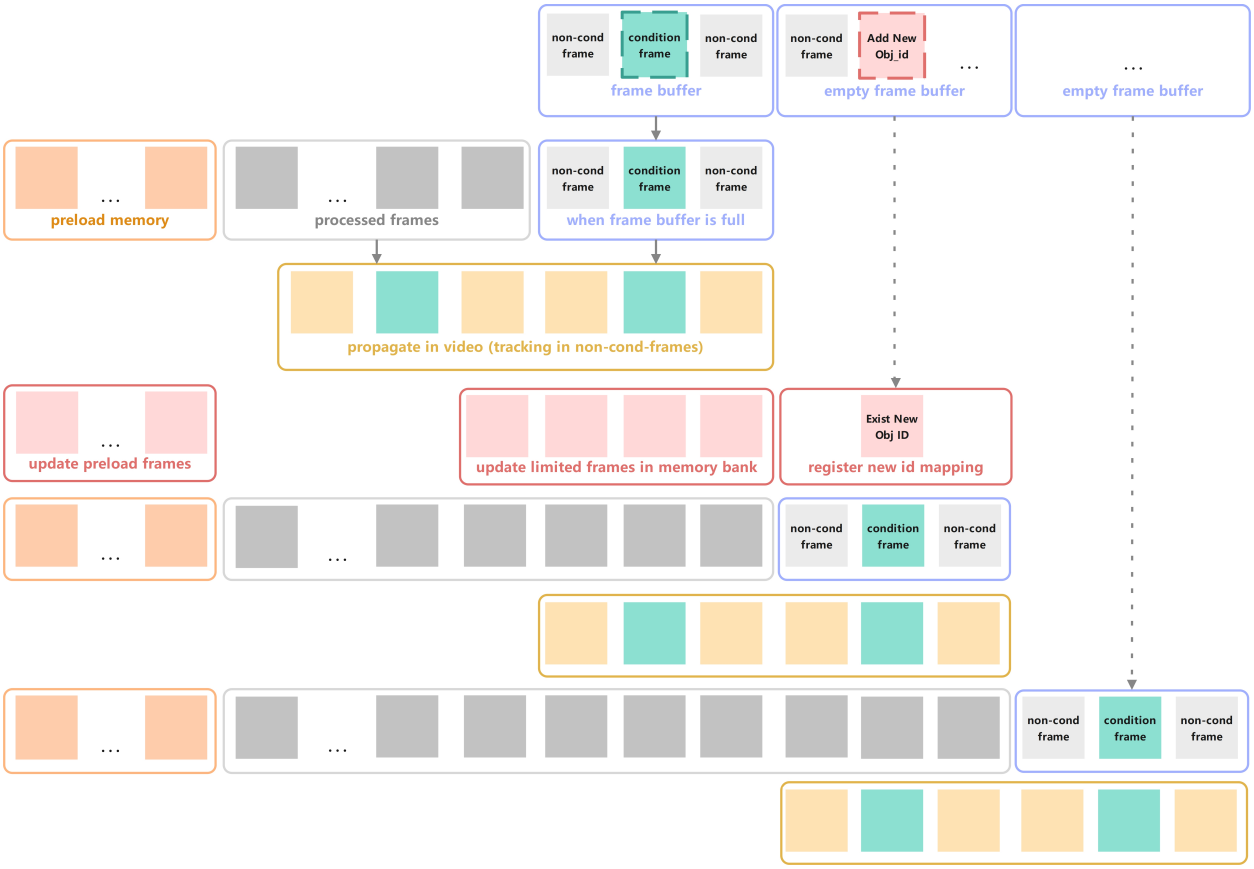

Figure 10: The optimized schematic diagram for online memory bank updates when new object IDs are input during the Det-SAM2 tracking process. By limiting the maximum number of frames updated in the Memory Bank and the number of condition frames used in Memory Attention calculation, while ensuring that the frames in the Preload Memory Bank are updated and involved in Memory Attention calculation, we can guarantee performance while reducing computational overhead.

3.7 GPU/CPU memory optimization

The official source code for SAM2[1] supports running on CPU, GPU, and NPU. However, since our own devices primarily use CPU and GPU, our optimizations mainly focus on GPU memory (VRAM) and system memory (RAM). In this section, we address the reduction of RAM and VRAM usage during the inference process in the Det-SAM2 architecture. We will introduce several interfaces reserved in the official source code, during which we will refer to many function names from the SAM2 source code.

Before optimization, the Det-SAM2 framework can infer approximately 200 video frames per 24GB of VRAM, with 6-7 segmented objects per frame. At this point, the RAM and VRAM usage increases linearly with the total number of frames in the video being processed.

1. First Optimization Interface: We will explore the first optimization interface reserved by the official source code, which is the `offload_video_to_cpu` parameter in the `SAM2VideoPredictor.init_state()` method. This parameter allows transferring the video frames in the memory bank (i.e., `inference_state["images"]`) from GPU VRAM to CPU RAM. At a video resolution of 1920x1080, this can reduce approximately 0.025GB of VRAM usage per frame, which means a reduction of 2.5GB of usage for every 100 frames.
2. Second Optimization Interface: Next, we will try the second optimization interface reserved by the official source code, which is the `offload_state_to_cpu` parameter in the `SAM2VideoPredictor.init_state()` method. The purpose of this parameter is to store large feature tensors, which do not require frequent computation, in CPU memory. However, in the Det-SAM2 framework we built, using this parameter does not directly save VRAM. Instead, it causes a misalignment between the generated segmentation masks and the frame indices.

   It was only after we set the tensor transfer parameter `non_blocking=False` at all locations where the `inference["storage_device"]` tensor that this interface began to function correctly.

   ```
   device = inference_state["storage_device"]
   tensor.to(device,non_blocking=False)
   ```

   When `offload_state_to_cpu=True`, the final effect after the fix, as mentioned in the official comments, is that VRAM usage is reduced while the inference time increases by approximately 22%.
3. Inspired by the discussion in the official repository issue[6], we hope to try clearing old frame data continuously (under the condition that the old frame data will no longer be used) in order to prevent the total memory usage from increasing indefinitely. To implement this functionality, we added the `release_old_frames()` method in the `SAM2VideoPredictor` class within `sam2.sam2_video_predictor`. This method allows setting a maximum number of frames to retain (`max_inference_state_frames`). Frames that are older than the maximum retention distance from the current frame will be considered as frames to be cleared.

   Therefore, to ensure that only frames that will no longer be used are cleared, `max_inference_state_frames` should be greater than the maximum propagation length (`max_frame_num_to_track`) in `propagate_in_video()`.

   The schematic diagram of the continuous old frame clearing process is shown in Figure 11, and the implementation of `release_old_frames()` can be found in Appendix A. With this approach, constant VRAM usage can be maintained during the inference process of infinitely long video frames.

   The diagram Figure 11 illustrates the case where the maximum number of retained frames is equal to the maximum propagation length (in the example, `max_inference_state_frames = max_frame_num_to_track = 4`). After each propagation (`propagate_in_video`), any processed frames that exceed the maximum retention limit (4 frames) will be released and cleared. In addition, the frames in the preload memory bank should never be released. Therefore, in this example (where the maximum number of retained frames equals the maximum propagation length) during the stable inference process of a long video:
   (a) The upper limit of VRAM usage is the data usage of `len(new_frames) + max_inference_state_frames + len(preload_memory)` frames.
   (b) The lower limit of VRAM usage is the data usage of `max_frame_num_to_track + len(preload_memory)` frames.
4. We unexpectedly found that the Memory Attention calculation in our pipeline generates a large number of intermediate variables that occupy VRAM and are not released in a timely manner. Therefore, we need to manually release the VRAM after the Memory Attention calculation. We discovered that this can significantly reduce the upper limit of VRAM usage in the Det-SAM2 pipeline. The specific implementation details can be found in Appendix B.

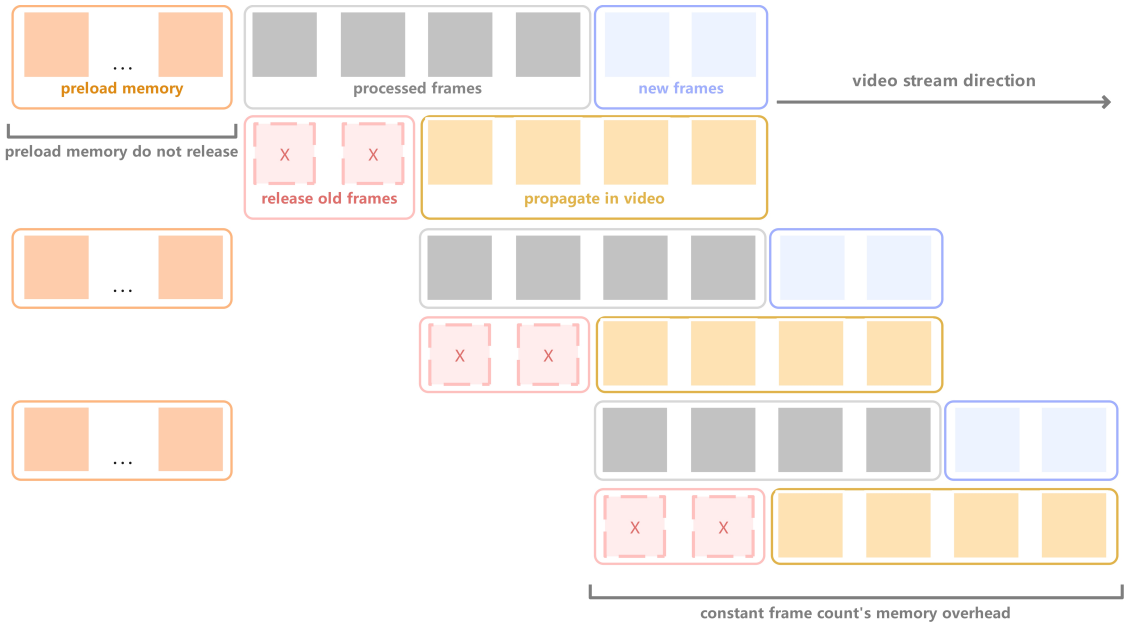

Figure 11: Schematic diagram of continuously releasing old frames to maintain constant VRAM usage.

5. Similarly, inspired by issue[7] in the official code repository, we tried storing the images in FP16 half precision instead of the original FP32. This saves approximately 0.007GB/frame of memory at a resolution of 1920x1080, with almost no loss in segmentation mask quality.
6. To further reduce the linear growth of memory usage while keeping VRAM usage constant, we aim to continuously clear old data, including clearing the cached video frames in `inference["images"]`. Although these frames have already been offloaded to CPU memory using the `offload_video_to_cpu=True` parameter, we still want to maintain a constant memory usage for this part (at a resolution of 1920x1080, 0.025GB/frame of memory usage grows linearly).

   To implement this functionality, we need to make four changes:

   (a) Modify the `SAM2VideoPredictor._get_image_feature()` method, which originally directly retrieves image frames from the corresponding frame index, to instead rely on an independent index mapping list (used to record the non-continuous index relationship in the video frame tensor) to fetch the corresponding frame tensor:

   ```
   target_idx = inference_state["images_idx"].index(frame_idx)
   image = inference_state["images"][target_idx].to(device).float().unsqueeze(0)
   ```

   (b) Implement the registration and updating of the `inference_state["images_idx"]` index mapping table in the `SAM2VideoPredictor.init_state()` and `update_state()` methods, respectively.

   (c) Add a function to clear old image frames in the `SAM2VideoPredictor.release_old_frames()` method. This will also synchronize the updates of `inference_state["images"]` and `inference_state["images_idx"]`.

   (d) Modify all references to `inference_state["num_frames"]` in the `SAM2VideoPredictor` class to ensure it always represents the total number of video frames that have been loaded historically, rather than the current total frame count (which may have deleted some old video frames).
7. We have successfully achieved constant VRAM usage, there remains a linear increase in memory consumption due to the ongoing clearing of the `inference["images"]` video frame cache. The question arises: How can we attain constant memory usage? In fact, the persistent linear growth in memory usage is primarily attributable to the segmentation result dictionary, `video_segment` that is continuously collected during the inference process:

   ```
   video_segments[out_frame_idx] = {
    out_obj_id: (out_mask_logits[i] > 0.0).cpu().numpy()
    for i, out_obj_id in enumerate(out_obj_ids)
   ```

```
}
```

By simply releasing the corresponding frame in the `video_segments` dictionary after processing the segmentation results for each frame in various downstream tasks, we can achieve constant memory usage:

```
import gc
...
# After your post-process in every frames
video_segments.pop(frame_idx, None)
gc.collect()
```

4 Experiments

We have integrated Det-SAM2 with a post-processing algorithm tailored to our specific business scenario (see Appendix C) to create a Det-SAM2-pipeline (see Appendix D). we present a visualization of the Det-SAM2 pipeline and provide an explanation of its functionality.

In the context of high-speed ball movement on a billiard table, Det-SAM2 can automatically infer long videos with the original accuracy of SAM2. It can accurately track the deformation and stretching of the ball (such as the ball with ID 9 in Figure 12), detect collisions between balls (such as the balls with IDs 9 and 17 in Figure 12), and accurately identifies when a ball bounces off the table's edge (such as the ball with ID 17 in Figure 12).

Figure 12: The visualized post-processing result of the Det-SAM2-pipeline in our self-implemented billiard scene example is shown in Figure 13. This image was derived from the segmentation masks predicted by SAM2, followed by post-processing. As seen, with the support of SAM2, even though the fast-moving balls are stretched in the camera frame, they are still perfectly captured.

Figure 13: Segmentation mask rendering of the Det-SAM2-pipeline in our self-implemented billiard scene example. The Det-SAM2 framework is able to maintain the original segmentation capabilities of SAM2 while operating autonomously.

5 Discussion

1. During the implementation process, we observed that the frequent generation of conditional prompts for SAM2 within the Det-SAM2 framework leads to an excessive dependence on the detection model. When certain frames lack prompts, SAM2 fails to predict segmentation results in scenarios where it could have normally performed well, with little to no additional interference. Therefore, it is necessary to carefully adjust the intervals at which the detection model is activated in custom scenarios to find the optimal parameters. We have developed an evaluation script specifically designed to assess the best combination of parameters in various example scenarios.

2. In our series of engineering implementations for resource optimization, we have constrained the capacity of the memory bank to store only a limited number of recent frame hints. This limitation will inevitably have an impact on videos with a large object association span. The positive aspect is that our detection branch can consistently provide hint information for the specified object categories throughout the entire video, serving as a continuous and fixed pseudo-memory source. However, to ensure optimal results, we still need to carefully adjust the maximum capacity of the memory bank in practical scenarios to determine the ideal value. This process can also be automated through a script designed to evaluate the best combination of parameters.

To maintain constant GPU memory usage, we have constrained the size of the memory bank. However, the question arises: is it possible to achieve constant memory usage while retaining comprehensive memory information? A potential direction for future research could involve rethinking the design of the memory bank by adopting architectures such as RWKV[8], which utilize weight-based state representations. This approach may facilitate multi-frame memory storage within a fixed parameter space.

3. One notable issue is the conceptual gap between the detection model and SAM2 during inference. The detection model outputs categories, and multiple objects of the same category can exist within a single frame. Conversely, SAM2

receives object IDs, and each object ID corresponds to only one object in a frame. A limitation of our Det-SAM2 is that it can only be used in scenarios where each category can only appear once in each frame. For example, in a pool game scenario, the detection model's output needs to treat each ball as a separate category, ensuring that each ball has a unique object ID when passed to SAM2. However, if our detection model only distinguishes between the cue ball and other balls, or only between solid and striped balls, SAM2 will face the problem of receiving multiple different prompts for the same object ID at different positions, which confuses SAM2.

We have not yet resolved the gap between the detection model's output categories and SAM2's concept of receiving object IDs in Det-SAM2. Some potential solutions involve adding engineering checks to manually assign unique and fixed SAM2 object IDs to different objects of the same detection category. However, this raises the challenge of how to differentiate between various objects of the same detection category to ensure that their IDs remain consistently matched.

6 Conclusion

This article presents the implementation process of Det-SAM2, a framework based on SAM2 that extends the system's capabilities without requiring manual interaction. We have implemented essential features for SAM2 to be applied in specific business scenarios, facilitating automatic SAM2 inference in long videos while maintaining constant GPU memory usage and overall memory efficiency. We anticipate that additional applications based on SAM2 will emerge in the near future.

Acknowledgments

We acknowledge the support provided by Motern AI.

Author Contributions

Zhiting Wang : Algorithm team leader. **Qiangong Zhou** : Core code contribution. **Zongyang Liu** : Project supervision.

References

[1] Nikhila Ravi, Valentin Gabeur, Yuan-Ting Hu, Ronghang Hu, Chaitanya Ryali, Tengyu Ma, Haitham Khedr, Roman Rädle, Chloe Rolland, Laura Gustafson, et al. Sam 2: Segment anything in images and videos. *arXiv preprint arXiv:2408.00714*, 2024.

[2] Alexander Kirillov, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alexander C Berg, Wan-Yen Lo, et al. Segment anything. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 4015–4026, 2023.

[3] Rejin Varghese and M Sambath. Yolov8: A novel object detection algorithm with enhanced performance and robustness. In *2024 International Conference on Advances in Data Engineering and Intelligent Computing Systems (ADICS)*, pages 1–6. IEEE, 2024.

[4] Glenn Jocher, Alex Stoken, Jirka Borovec, Liu Changyu, Adam Hogan, Laurentiu Diaconu, Francisco Ingham, Jake Poznanski, Jiacong Fang, Lijun Yu, et al. ultralytics/yolov5: v3. 1-bug fixes and performance improvements. *Zenodo*, 2020.

[5] petercsjTW. Issue comment on sam2 github repository: Reuse the memory bank for another video. `https://github.com/facebookresearch/sam2/issues/210`, 2024. Accessed: 2024-8-13.

[6] heyoeyo. Issue comment on sam2 github repository: Are there any method for reducing gpu memory overhead? `https://github.com/facebookresearch/sam2/issues/196#issuecomment-2286352777`, 2024. Accessed: 2024-8-11.

[7] ronghanghu. Issue comment on sam2 github repository: Are there any method for reducing gpu memory overhead? `https://github.com/facebookresearch/sam2/issues/196#issuecomment-2475114783`, 2024. Accessed: 2024-8-11.

[8] Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Michael Chung, Matteo Grella, et al. Rwkv: Reinventing rnns for the transformer era. *arXiv preprint arXiv:2305.13048*, 2023.

A Release Old Frames

In the `sam2.sam2_video_predictor` module, the `SAM2VideoPredictor.release_old_frames()` method clears the old frames and specifically releases the non-condition frames in `output_dict` and `output_dict_per_obj`, as well as the condition frames in `output_dict`, `output_dict_per_obj`, and `consolidated_frame_inds`.

```
def release_old_frames():
    ...
    # delate old non_cond_frames
 inference_state['output_dict']['non_cond_frame_outputs'].pop(old_idx)
 for obj in inference_state['output_dict_per_obj'].keys():
  inference_state['output_dict_per_obj'][obj]['non_cond_frame_outputs'].pop(old_idx)
 # delate old cond_framse
 inference_state['output_dict']['cond_frame_outputs'].pop(old_idx)
 inference_state['consolidated_frame_inds']['cond_frame_outputs'].discard(old_idx)
 for obj in inference_state['output_dict_per_obj'].keys():
  inference_state['output_dict_per_obj'][obj]['cond_frame_outputs'].pop(old_idx)
    ...
```

B Release GPU memory after Memory Attention

In the `sam2.modeling.sam2_base` module, we add a manual memory release operation after the Memory Attention calculation in the `SAM2Base._prepare_memory_conditioned_features()` method.

```
def _prepare_memory_conditioned_features():
 ...
    pix_feat_with_mem = self.memory_attention(
        curr=current_vision_feats,
        curr_pos=current_vision_pos_embeds,
        memory=memory,
        memory_pos=memory_pos_embed,
        num_obj_ptr_tokens=num_obj_ptr_tokens,
    )
    # Add release GPU memory
    torch.cuda.empty_cache()
    ...
```

C Post-processing Example

As shown in Figure 1, post-processing is a necessary step for Det-SAM2 to move towards higher-level applications. We have implemented a post-processing example in the billiard scene to demonstrate the potential of our Det-SAM2 in practical applications. Our post-processing example primarily designs three event detection algorithms for billiard scenes, used to determine: goals, ball-to-ball collisions, and ball rebounds off the table edges.

Specifically, in `postprocess_det_sam2.py`, we first calculate the centroid of each segmentation mask (i.e., the position coordinates of each ball) and the velocity vector of the ball between every two frames based on the masks. Using the position coordinates and velocity vectors as the foundation, we perform mid-level event detection, such as goals, collisions, and pocket bounces.

C.1 Goal Detection

First, obtain the positions of the six pockets from the SAM2 inference backbone's detection model, assign names to the six pockets, and determine which position corresponds to which pocket.

During the traversal of each frame, the following conditions are checked:

1. The ball's position in the previous frame is near a pocket, and the ball disappears in the current frame.
2. The ball's velocity in the previous frame points towards the pocket.

If both conditions are satisfied, it is determined that the ball has entered the target pocket.

Correction Mechanism: If the same ball is detected entering a pocket again in subsequent frames, the latest goal information will overwrite the previous record.

C.2 Ball Collision Detection

During the traversal of each frame, collision detection is triggered when a ball's velocity vector undergoes a significant change (exceeding a defined threshold). The following conditions are checked:

1. Identify the ball that might have collided with the current ball by analyzing the velocity vectors before and after the event:
   (a) Before the collision, the two balls are moving towards each other.
   (b) After the collision, the velocities of the two balls change significantly, and their accelerations exhibit correlations (e.g., the introduction of components indicating they are moving away from each other).
2. Determine if the potential collision ball is near the current ball.

If both conditions are satisfied, it is determined that a collision occurred between the two balls.

Correction Mechanism: If the same frame is reevaluated later and yields a different result, the new judgment will overwrite the previous information.

Note: Collision detection requires acceleration calculations, which necessitate data from the current frame and the two preceding frames, totaling three frames of information.

C.3 Table Edge Rebound Detection

First, extract the four valid boundaries of the table (top, bottom, left, right) from the coordinates of the six pockets. Shrink these boundaries inward to create buffer zones near each edge, which are used to trigger rebound detection.

During the traversal of each frame, when a ball enters a buffer zone near the boundaries, the boundary position (top, bottom, left, or right) is recorded, and rebound detection is triggered based on the following conditions:

1. Check if the ball was moving toward the corresponding boundary before the rebound (in the previous frame).
2. Check if the ball is moving away from the corresponding boundary after the rebound (in the current frame).
3. Verify whether the velocity component perpendicular to the boundary has essentially reversed direction. If not, check if the velocity component parallel to the boundary remains approximately consistent.

If conditions 1, 2, and 3 are all satisfied, it is determined that the ball rebounded off the corresponding boundary. If condition 1 is satisfied but conditions 2 and 3 are not, further checks are performed to determine if the irregular behavior is due to the ball hitting a curved surface near a pocket:

1. Check if the ball is near a pocket.
2. Determine if the velocity vector has changed significantly between the current frame and the previous frame (indicating a possible external collision).
3. Confirm that the velocity vector in the previous frame is not directed toward any other ball (to rule out ball-to-ball collisions near the pocket).
4. Verify that there are no collisions involving the ball in the current frame (using the collision results dictionary).

If conditions 4, 5, 6, and 7 are all satisfied, it is also determined that the ball rebounded off the corresponding boundary.

Correction Mechanism: If the same frame is reevaluated later with inconsistent results, the new judgment will overwrite the previous one.

Note: Rebound detection requires velocity calculations, which depend on data from the current frame and the two preceding frames, totaling three frames of information.

D Det-SAM2-pipeline

The Det-SAM2-pipeline is a complete workflow that integrates Det-SAM2 with post-processing.

The script `Det-SAM2_pipeline.py` utilizes the video inference backbone class from `det_sam2_RT.py` and the post-processing class from `postprocess_det_sam2.py`, combining them within the `DetSAM2Pipeline.inference()` function.

During the execution of `DetSAM2Pipeline.inference()`, the SAM2 video inference backbone and the post-processing components each operate in separate threads, enabling an asynchronous and parallel workflow:

1. Main Inference Thread: Responsible for reading data frame by frame from the video stream and performing detection and segmentation inference.
   (a) Reads each frame from the video stream as input from the video source.
   (b) Passes the frames into the Det-SAM2 inference framework, where the detection model provides conditional prompts for SAM2 to perform segmentation and segmentation correction in a real-time video stream.
   (c) Stores the segmentation results of each inference (`propagate_in_video`) into the inference result cache `video_segments`. Newly added inference results are also pushed to the post-processing queue (`frames_queue`).
   (d) Triggers the post-processing thread once the required settings (e.g., pocket coordinates, table boundaries) are collected, ensuring the post-processing thread is activated.
2. Post-Processing Thread: Handles the segmentation results pushed by the main inference thread in parallel, performing further object tracking and state analysis.
   (a) Monitors the post-processing queue (`frames_queue`), and starts processing as soon as new inference results are available. It processes all frames sequentially (may reprocess previously processed frames) but does not skip frames to directly process later ones.
   (b) Uses the segmentation results retrieved from `frames_queue` to calculate the ball positions and velocity vectors.
   (c) Performs goal detection starting from frame 2.
   (d) Performs collision detection starting from frame 3.
   (e) Performs rebound detection starting from frame 3.

This two-thread asynchronous parallel workflow ensures the Det-SAM2 system is efficient, accurate, and capable of real-time processing.