Latency Analysis and Optimization of Alpamayo 1 via Efficient Trajectory Generation

Yunseong Jeon*, Namcheol Lee†, Yoonsu Lee*, Jangwoon Park†, Sol Ahn*, Jong-Chan Kim*, Seongsoo Hong†
* Graduate School of Automobile and Mobility, Kookmin University, Seoul, Republic of Korea
† Department of Electrical and Computer Engineering, Seoul National University, Seoul, Republic of Korea
{kb0316, dldbs0319, solahn00, jongchank}@kookmin.ac.kr, {nclee, jwpark, sshong}@redwood.snu.ac.kr

***Abstract*—Reasoning-based end-to-end (E2E) autonomous driving has recently emerged as a promising approach to improving the interpretability of driving decisions as it can generate human-readable reasoning together with predicted trajectories. Such approaches commonly generate multiple trajectories to capture diverse future behaviors, and they fall into two categories: (1) multi-reasoning, where one reasoning sequence is generated per trajectory, and (2) single-reasoning, where a single reasoning is shared across all trajectories. The former offers richer diversity at the cost of redundant computation, while the latter is more efficient but is often assumed to sacrifice diversity. Alpamayo 1, a representative system, adopts the multi-reasoning approach and achieves competitive trajectory prediction performance. However, the efficiency of this design remains largely unexplored, making it a well-motivated subject for investigation. In this paper, we systematically analyze and improve Alpamayo 1 in two ways. First, we reduce inference latency while preserving trajectory diversity by redesigning Alpamayo 1 into a single-reasoning system. Through extensive experiments, we find that replacing multi-reasoning with single-reasoning does not meaningfully degrade trajectory diversity. Second, we accelerate diffusion-based action generation by eliminating inter-block overhead arising from unnecessary copy operations and inefficient kernel execution. Through closed-loop and open-loop experiments, we validate both optimizations, demonstrating a 69.23% reduction in inference latency while maintaining trajectory diversity and prediction quality. These results highlight the importance of jointly analyzing system architecture and runtime execution to improve the efficiency of reasoning-based E2E AD systems.**

I. Introduction

End-to-end (E2E) autonomous driving has recently gained significant attention as a paradigm that directly maps sensor observations to driving actions or trajectories using deep neural networks (DNNs) [1]–[6]. While this approach simplifies the traditional modular approaches, it often suffers from limited interpretability, making it difficult to understand the rationale behind driving decisions. To address this limitation, *reasoning-based* approaches have recently emerged as a promising direction [7]–[21]. These models augment trajectory generation with large language model (LLM)-based reasoning, enabling better transparency and diagnosability.

Since driving scenarios often admit multiple plausible futures, reasoning-based models commonly generate multiple candidate trajectories to capture diverse future driving behaviors. In this multi-trajectory generation, the relationship between reasoning generation and trajectory generation gives rise to two distinct design paradigms. The first is the *multi-reasoning* approach, where a separate reasoning is generated for each predicted trajectory [7]–[17]. This design allows each trajectory to be conditioned on its own reasoning, potentially improving behavioral diversity. However, it also introduces significant computational redundancy because reasoning has to be repeatedly generated for each trajectory. The second paradigm is the *single-reasoning* approach, in which a single reasoning is shared across all predicted trajectories [18]–[21]. While this design is computationally more efficient, it is commonly assumed to reduce trajectory diversity because all trajectories rely on the same reasoning. Nonetheless, since trajectory generation is inherently stochastic, the single-reasoning approach can still capture diverse behaviors.

Among recent reasoning-based autonomous driving models, *Alpamayo 1* from NVIDIA (hereinafter Alpamayo) is a representative vision-language-action (VLA) model that integrates vision-language model (VLM)-based reasoning with diffusion-based action (i.e., steering and acceleration) generation (actions can be converted into trajectories) and achieves competitive trajectory prediction performance [7]. Despite its effectiveness, the efficiency of this model and its inference architecture remains largely unexplored. In particular, Alpamayo's multi-reasoning design together with iterative diffusion-based action generation introduces substantial latency overhead as the number of trajectories grows, which can limit its practicality in real-time driving systems. This motivates a careful analysis of the architectural and runtime factors that contribute to the latency of Alpamayo.

In this study, we perform a comprehensive analysis of Alpamayo's runtime architecture and propose complementary optimizations to significantly reduce its inference latency while preserving trajectory diversity and prediction quality. We first conduct a detailed architectural dissection of Alpamayo, examining each component and its role in the inference pipeline. Based on this dissection, we perform a fine-grained latency analysis that reveals the following two dominant bottlenecks. First, the multi-reasoning design requires a separate reasoning to be generated for each trajectory, resulting in inference latency that scales linearly with the number of trajectories. Second, the diffusion-based action generation incurs substantial latency overhead during denoising steps. It is caused by unnecessary memory copies and reallocations for the dynamic KV cache management and by the repeated fine-grained GPU kernel launches from CPU that prevents continuous GPU kernel executions.

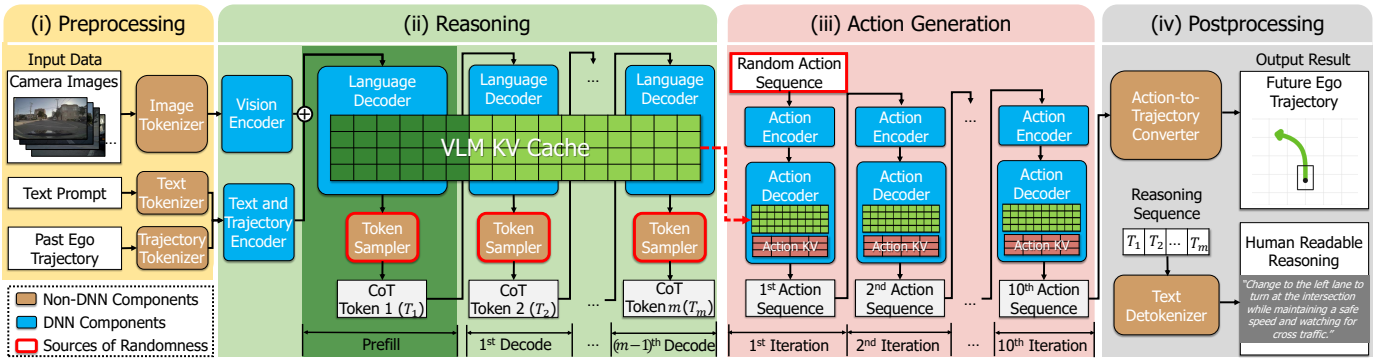


Fig. 1: Inference pipeline of Alpamayo 1 (hereinafter Alpamayo)

Guided by these findings, we propose two targeted optimizations. First, we redesign Alpamayo into a single-reasoning architecture, where a single reasoning conditions all trajectory generation, eliminating redundant computation. Through extensive experiments, we demonstrate that this redesign does not meaningfully degrade trajectory diversity and prediction quality, challenging the widely held assumption that multi-reasoning is necessary for diverse predictions. Second, we eliminate the identified latency overhead in the action generation. For that, we implement a static KV cache management scheme that preallocates a static KV buffer and reuses it throughout the diffusion iterations. This approach is feasible because we assume a static periodic application (i.e., autonomous driving), where its maximum KV cache size can be precisely estimated by profiling the KV cache growth pattern offline. Also, we apply the CUDA graph capture technique to minimize the GPU kernel launch overhead by recording a sequence of GPU kernel launch commands and just replaying it without additional launch overhead. This can be possible since we are using a static KV cache with a known KV buffer address that does not change throughout the action generation process.

We evaluate the proposed optimizations through open-loop experiments on the NVIDIA Physical AI Dataset [22] and closed-loop experiments on the AlpaSim autonomous driving simulator [23]. The results demonstrate up to a 69.23% reduction in inference latency while maintaining comparable trajectory diversity and prediction quality.

The contributions of this study can be summarized as:

- We present a detailed breakdown of Alpamayo's inference architecture and a per-component latency analysis obtained by profiling its runtime execution.
- We compare the single-reasoning and multi-reasoning approaches for multi-trajectory generation and redesign Alpamayo into a single-reasoning system.
- We reduce the inference latency by removing redundant memory copy overhead and kernel launch delays during diffusion-based action generation.

II. Alpamayo Overview

A. Overall Architecture

The official description of Alpamayo can be found in [7]. However, it focuses only on the neural architecture and the training method of the model itself without describing the runtime inference architecture. In this regard, we analyze the runtime architecture of Alpamayo based on the open source inference code officially released by NVIDIA, which comprises 3,462 lines of Python source code [24]. The codebase is based on PyTorch [25] and the Hugging Face Transformers library [26].

Alpamayo is a VLA model composed of several DNN models, among which the following three transformer-based models are the most central:

- **Vision encoder** converts input image tokens into visual token embeddings, which are multi-dimensional vector representations of image tokens that can be understood by language models in addition to text prompts (composed of 27 transformer blocks).
- **Language decoder** does the actual language-based reasoning to produce so called the chain-of-thought (CoT) tokens such that the reasoning can be understood by humans while the reasoning is correctly reflected in the trajectory generation afterwards (36 transformer blocks).
- **Action decoder** generates a sequence of actions (i.e., acceleration and steering) to be executed by the ego vehicle for the next time steps, which eventually can be interpreted as a 2-dimensional trajectory for visualization and evaluation (36 transformer blocks).

In general, an LLM is technically a language decoder that produces output words (i.e., tokens) in an autoregressive manner from given input prompts. An LLM augmented with a vision encoder is called a VLM that can interpret images in addition to languages. When a VLM is further coupled with an action decoder that takes the VLM's output or intermediate activations as input, just like Alpamayo, they are collectively recognized as a VLA model.

Fig. 1 depicts the inference pipeline of Alpamayo, which is composed of four modules: (i) preprocessing, (ii) reasoning, (iii) action generation, and (iv) postprocessing. In each

module, the blue components are DNNs, while the brown components are not. A detailed explanation of each module is provided in Section II-B.

Inputs. As inputs, camera images, text prompt, and past ego trajectory are given to the system. Camera images include four most recent sets of images from four cameras (i.e., front, front-zoom, left, and right) at a 10 Hz sampling rate. The text prompt consists of a *system prompt* and a *user prompt*, where the system prompt defines the identity (or *persona*) of the language model. Alpamayo specifically uses the following system prompt:

> *You are a driving assistant that generates safe and accurate actions.*

The following user prompt is also used:

> *Output the chain-of-thought reasoning of the driving process, then output the future trajectory.*

The ego trajectory history is represented as a sequence of 16 ego states sampled at 10 Hz over the past 1.6 seconds, expressed in the vehicle-centric coordinate frame defined at the current time step. During inference, the inputs are given in the order of system prompt, camera images, user prompt, and past ego trajectory.

Outputs. As outputs of the system, N reasoning sequences and their corresponding future trajectories are generated. By default, Alpamayo generates a single trajectory (N=1), which can be adjusted by modifying a hyperparameter. Each trajectory is converted from an action sequence produced by the action decoder. An action sequence is represented as $\boldsymbol{a} = \{(a^i, k^i)\}_{i=1}^{64}$, where a^i and k^i denote the acceleration and curvature at the i-th future timestep, respectively, with each timestep corresponding to 0.1 seconds (10 Hz). The resulting ego trajectory for the next 6.4 seconds is expressed as $\boldsymbol{\tau} = \{(x^i, y^i, \theta^i_{\text{yaw}})\}_{i=1}^{64}$, where (x^i, y^i) denotes the two-dimensional position in the ego vehicle's coordinate and θ^i_{yaw} denotes the heading angle. Additionally, human readable reasoning messages are generated from the reasoning sequences from the language decoder.

B. Detailed Inference Pipeline

As illustrated in Fig. 1, the inference pipeline of Alpamayo proceeds through four modules in sequence: preprocessing, reasoning (with vision encoder and language decoder), action generation (with action decoder), and postprocessing. We first describe the single-trajectory case in this section and discuss the multi-trajectory generation in Section II-C.

(i) Preprocessing: Shown as the yellow rectangle in Fig. 1, it consists of three components: an image tokenizer, a text tokenizer, and a trajectory tokenizer. During inference, camera images are tokenized by the image tokenizer, which follows the image preprocessor of Qwen3-VL [27] and splits each image into patches of 14×14 pixels. The text prompt is processed by the text tokenizer, likewise identical to that of Qwen3-VL. Past ego trajectory is handled by the trajectory tokenizer, which extends the text tokenizer with additional vocabularies for discretized trajectory representation, allowing trajectory information to occupy the same token space as textual inputs. The resulting tokens from the image tokenizer are passed to the vision encoder whereas the text and trajectory tokens are concatenated and fed to the text and trajectory encoder.

(ii) Reasoning: Shown as the green rectangle in Fig. 1, it comprises a vision encoder, a text and trajectory encoder, a language decoder, and a token sampler. The vision encoder, based on Qwen3-VL and composed of a three-dimensional convolutional layer followed by a Vision Transformer with 27 transformer blocks, maps image tokens into embeddings, which are multi-dimensional vectors. Text and trajectory tokens are simultaneously converted to embeddings by the text and trajectory encoder, which is implemented as a lookup table. These embeddings are then processed by the language decoder, a decoder-only transformer with 36 transformer blocks and a linear projection head derived from the Qwen3-VL language model. The token sampler then selects the next token from the probability distribution derived from the logits produced by the language decoder. While the official release [7] refers to the language decoder as the Cosmos-Reason 1 [28] backbone, the two may not share identical weights, as Cosmos-Reason 1 is built on Qwen2.5-VL [29] whereas the open-source release of Alpamayo is built on Qwen3-VL.

The language decoder operates in two distinct phases: *prefill* and *decode*. In the prefill phase, all input embeddings are processed in parallel through the 36 transformer blocks, building an initial understanding of the full input context. In the decode phase, CoT reasoning tokens are generated autoregressively via the language decoder and the token sampler. Each new token is conditioned on all previously generated tokens until a termination token is produced or the maximum generation length is reached. The first token is generated by passing the last hidden state from the prefill phase through the linear projection head and the token sampler. Throughout both phases, each transformer block accumulates KV caches, internal buffers that store intermediate computation results for all previously processed tokens, allowing the model to avoid redundant recomputation when generating subsequent tokens. Upon producing m CoT tokens $\{T_1, T_2, \cdots, T_m\}$, these KV caches serve as the contextual representation that conditions the downstream diffusion-based action generation module. One important observation is that there is no separate interface between the reasoning module and the action generation module. The resulting KV cache content from the reasoning module plays the role of input to the action generation module.

(iii) Action generation: Shown as the red rectangle in Fig. 1, it consists of two components: an action encoder and an action decoder. The module begins with a randomly initialized action sequence and refines it over 10 iterations. In each iteration, the action encoder, a sinusoidal positional

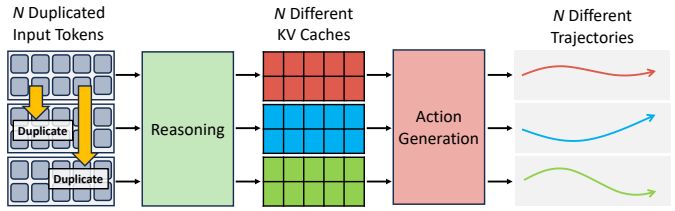
encoding followed by a multi-layer perceptron, maps the current action sequence to action embeddings. These embeddings are then processed by the action decoder, which shares the same transformer architecture as the language decoder but with a different hidden dimension, compatible key and value dimensions, and a linear projection head that maps the output to the action space. Within each transformer block of the action decoder, attention is computed from two distinct sources of keys and values: action-derived key-value pairs computed from the action embeddings themselves, and the KV caches passed from the language decoder. This dual-source attention allows each refinement step to simultaneously integrate action-level context and the full scene understanding encoded during reasoning. Unlike the language decoder, the action decoder processes all action embeddings in a single parallel pass, after which the linear projection head converts each output embedding into an incremental action update. This update, scaled by 0.1, is added to the current action sequence, and the result becomes the input for the next iteration. After all 10 iterations, the final action sequence is passed to the postprocessing module.

(iv) Postprocessing: Shown as the gray rounded rectangle in Fig. 1, it converts generated outputs into interpretable results. It consists of two components: a text detokenizer and an action-to-trajectory converter. The text detokenizer, the inverse of the text tokenizer in the preprocessing module, maps the reasoning token sequence back to human-readable text. The action-to-trajectory converter then transforms the final action sequence into a future ego trajectory based on the unicycle dynamics.

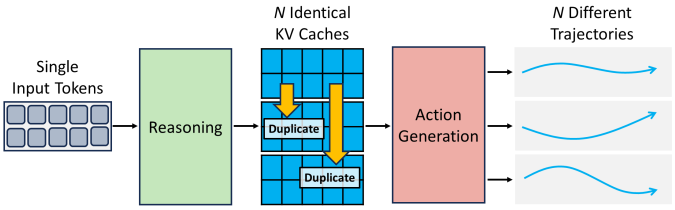
Among the modules, the reasoning and action generation modules contain transformer-based models, whereas the preprocessing and postprocessing modules involve no learned components. For simplicity, positional encodings, which inject token position information into the embeddings, are omitted from Fig. 1. Specifically, the vision encoder and the language decoder apply positional encodings internally, while the action decoder receives its position encodings externally, computed by the language decoder. DeepStack [30], which is applied to the language decoder, is also omitted because it focuses on improving visual understanding during the prefill phase, which is orthogonal to our proposed solution.

C. Generating Multiple Trajectories

Multi-reasoning (N:N). Alpamayo uses the multi-reasoning approach for generating multiple trajectories. As illustrated in Fig. 2a, multi-reasoning extends single-trajectory generation to N multi-trajectory generation through batched inference, not iterative inference. The input tokens are replicated to form a batch of size N, and the reasoning generation module processes the copies in a batched manner, producing N distinct reasoning sequences and their associated KV caches. The action generation module then operates on a batch of N randomly initialized action sequences, each conditioned on its own KV cache, yielding N trajectories. Trajectory diversity in this approach arises from two sources: stochastic token sampling during reasoning generation, which produces different reasoning sequences and therefore different KV caches, and random initialization of the action sequences in the action generation module.



(a) Multi-reasoning architecture



(b) Single-reasoning architecture

Fig. 2: Comparison of multi-reasoning and single-reasoning architectures

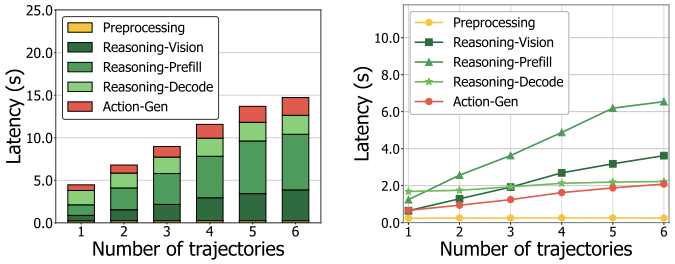
Single-reasoning (1:N). A possible alternative is to use the single-reasoning approach. As illustrated in Fig. 2b, single-reasoning generates a single reasoning sequence and its associated KV cache from the input tokens, then shares this single KV cache across a batch of N randomly initialized action sequences. The action generation module thus produces N trajectories conditioned on the same contextual representation. Trajectory diversity in this approach relies solely on the random initialization of action sequences, since the reasoning sequence and its KV cache are identical for all N trajectories. The diversity advantage of multi-reasoning over single-reasoning therefore hinges on how strongly variations in the reasoning sequence influence the downstream action generation. If this influence is limited, the stochasticity of action sequence initialization alone may be sufficient to achieve comparable trajectory diversity.

III. Architecture Redesign by Latency Analysis

A. Profiling Setup

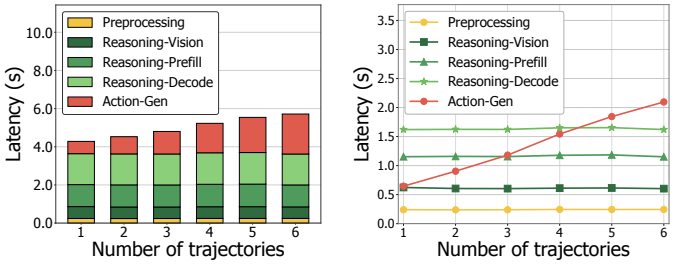
For the latency component profiling of Alpamayo, we use an NVIDIA DGX Spark machine with a GB10 Grace Blackwell processor and 128 GB of integrated memory. As the operating system, we use NVIDIA DGX OS 7, which is based on Ubuntu 24.04 with preinstalled GPU drivers and CUDA runtime. As an evaluation dataset, a random scenario with 16 camera images is selected from NVIDIA Physical AI Dataset. Each reported result represents the average of 100 measurements. For profiling, we modified the Alpamayo inference code and the Hugging Face Transformers library.

Based on the profiling results, Alpamayo's inference latency can be analyzed by five major latency components, denoted as follows:



(a) Latency breakdown (b) Per-component latency trends

Fig. 3: Latency analysis of multi-reasoning architecture



(a) Latency breakdown (b) Per-component latency trends

Fig. 4: Latency analysis of single-reasoning architecture

- **Preprocessing:** Latency caused by the preprocessing module, which is mostly for input token generations from inputs.
- **Reasoning-Vision:** Latency caused by the vision encoder in the reasoning module, which is mostly about executing ViT-based transformer blocks for image encoding.
- **Reasoning-Prefill:** Latency caused by the prefill phase by the language decoder in the reasoning module, which populates KV caches for subsequent token generations.
- **Reasoning-Decode:** Latency caused by the autoregressive CoT token generations during the decode phase in the reasoning module; this latency grows with the number of generated tokens.
- **Action-Gen:** Latency caused by the iterative refinement of action sequences by the action decoder in the action generation module, which runs for 10 iterations.

Since the postprocessing module's latency is negligible without visible impact on the inference latency, we do not separately report the postprocessing latency. We explain the profiling results for the multi-reasoning and single-reasoning cases in Section III-B and Section III-C, respectively. Then Section III-D demonstrates the relationship between the action generation latency and the number of generated CoT tokens.

B. Alpamayo's Multi-Reasoning Architecture

Fig. 3a presents the per-component latency as a stacked bar graph for trajectory counts ranging from 1 to 6 using Alpamayo's default *multi-reasoning* architecture. Fig. 3b presents the same results separated by latency component to highlight the average latency growth slope per component.

The preprocessing latency remains constant at around 0.25 s regardless of the number of trajectories, indicating that input

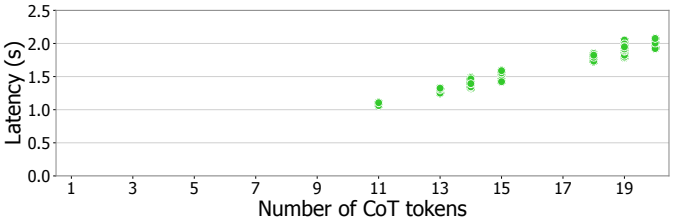


Fig. 5: Relationship between number of CoT tokens and decoding latency

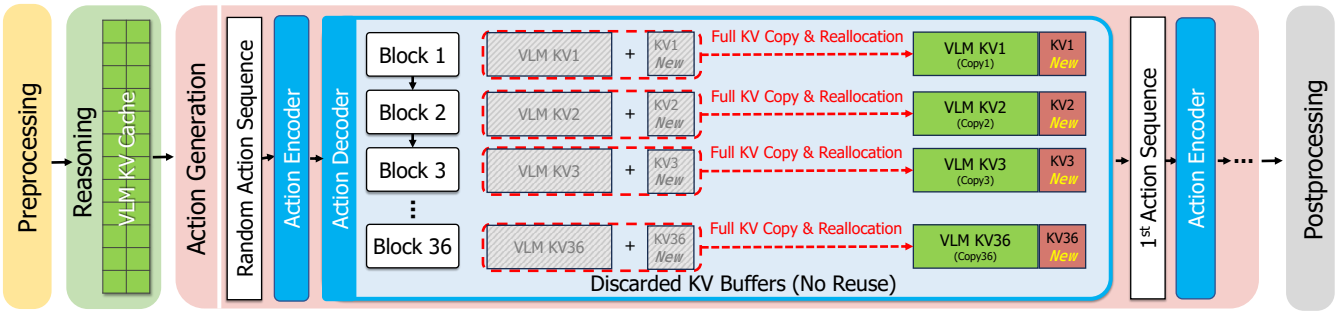
duplication occurs after the preprocessing module. In contrast, the reasoning latency increases almost linearly with the number of trajectories, as the duplicated inputs are processed in a batched manner, scaling the computational workload proportionally. However, by looking at Fig. 3b, we can notice that the slopes are different for each latency component. More specifically, Reasoning-Vision and Reasoning-Prefill exhibit similar average latency-growth slopes of 5.63 and 5.28, respectively. In contrast, the corresponding slope for Reasoning-Decode is substantially lower at 1.32. This discrepancy indicates the different computational intensity of each component, meaning Reasoning-Decode requires far less computation than Reasoning-Vision and Reasoning-Prefill. The average latency-growth slope of Action-Gen is 3.15, indicating that its computational density lies between those of Reasoning-Vision and Reasoning-Decode.

Discussion. Note that even in a very powerful DGX Spark machine, the inference latency for the single trajectory generation is about 4 seconds, which is not practical for real-time processing. However, this does not mean that Alpamayo is not useful in real-world autonomous driving. VLA-based models are often concurrently used in combination with lightweight E2E models for safety regulation purposes [11], [12], where high inference latencies are acceptable.

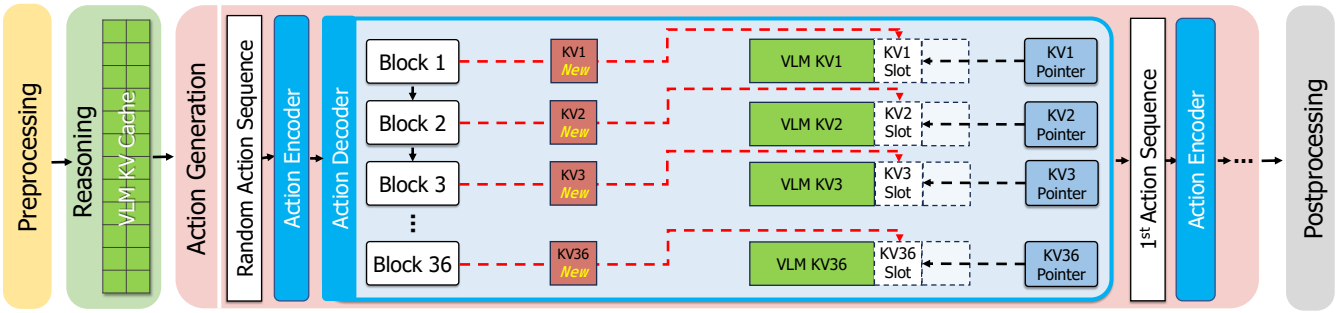
C. Our Single-Reasoning Architecture

To overcome the limitations of the multi-reasoning architecture, we modify Alpamayo to adopt a *single-reasoning* architecture, eliminating input duplication so that the reasoning module generates a single reasoning sequence shared across all trajectories. To generate multiple trajectories, the action generation module performs batched inference using replicated KV caches derived from the single reasoning sequence. Despite using identical KV caches, diverse trajectories can still be generated owing to the random initialization of the action sequence at the beginning of the action generation process.

Fig. 4 shows the profiling results in our single-reasoning implementation, where it begins with the same latency as the multi-reasoning case when the number of trajectories is 1. As the number of trajectories increases, all the latency components other than the action generation latency are constant, meaning that there is no additional computation overhead affected by the number of trajectories. Only the action generation latency scales by 3.15, which is the same as the multi-reasoning case. By looking at Fig. 4a, the proportion



(a) Baseline action generation architecture with *dynamic* KV cache management scheme



(b) Our optimized action generation architecture with *static* KV cache management scheme

Fig. 6: Optimizing action generation

of action generation is only 15.06 % when the number of trajectories is 1. However, when the number of trajectories is 6, the proportion reaches 33.67 %, which is significant within the overall latency.

Discussion. It is difficult to make a direct comparison between the multi-reasoning architecture and the single-reasoning architecture across all evaluation aspects. However, it is obvious that latency overhead caused by the multi-reasoning architecture is critical when we need a large number of trajectories. Thus, on the condition that the trajectory diversity and the resulting driving performance remains with little changes, the single-reasoning architecture can be a better solution.

D. Decoding Latency

Fig. 5 shows the decoding latency for varying numbers of generated CoT tokens. The figure shows that decoding latency increases linearly with the number of generated tokens since the tokens per second (TPS) performance is invariant with a given language model on a fixed hardware platform, which is 10.12 TPS in this case. The maximum token generation length of Alpamayo is 256 by default, but in practice it generates between 11 and 20 CoT tokens, limiting the latency variation to a narrow range. To guarantee a hard decode latency, we may impose a less number of output tokens than the default one. Nonetheless, limiting output tokens may meaningfully affect the reasoning capability of language models [31], [32]. We therefore maintain the default limit, but we have not experienced more than 20 CoT tokens yet. This output token limit issue is outside the scope of this study.

IV. Optimizing Action Generation

A. Baseline Action Generation Architecture

Fig. 6a shows Alpamayo's baseline action generation architecture. It begins with a random action sequence, which is refined with 10 diffusion iterations. Each diffusion iteration is composed of 36 transformer blocks. Action transformer blocks reference the KV cache generated by their corresponding transformer blocks in the reasoning module to incorporate reasoning context into action generation. At the same time, they produce new KV entries that are appended to the corresponding KV caches. This block-wise KV cache referencing and updating process is repeated across all transformer blocks. After Block 36, a new diffusion iteration begins using the reasoning module's KV cache together with the action sequence output from the previous iteration. This execution pattern continues until the final diffusion iteration.

Alpamayo adopts a *dynamic* KV cache management scheme, where each transformer block copies the previously accumulated KV cache (full KV) concatenated with newly generated KV tensors to a newly allocated buffer. This dynamic scheme is commonly used in conventional LLM applications like chatbots, as the output sequence length is non-deterministic, and thus the maximum KV cache size cannot be determined in advance. While dynamically expanding the KV cache is a scalable solution in such cases, it introduces significant memory copy and reallocation overhead.

B. Optimization 1: Static KV Cache Management

To address the inefficiency of dynamic KV cache management, we leverage a key characteristic of the action generation

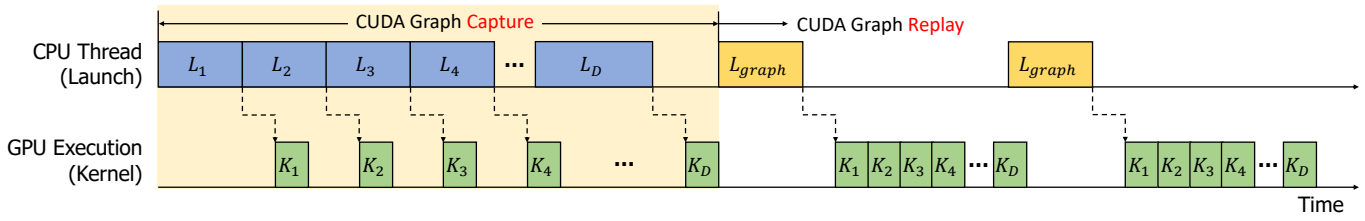


Fig. 7: CUDA graph capture

module: its memory usage becomes predictable after the reasoning stage. Although the KV cache produced by the VLM cannot be determined in advance, it remains unchanged during diffusion. Furthermore, the KV entries generated during diffusion are identical in size across iterations. Specifically, the KV cache received from the language decoder occupies approximately 454.3 MB for a single trajectory when assuming 20 CoT tokens. During action generation, the KV cache increases by 9.44 MB, reaching a maximum size of 463.74 MB of KV cache. Based on this observation, we implement a *static* KV cache management scheme that preallocates the maximum required memory in advance and updates the KV cache via in-place writes. Specifically, we maintain a KV pointer that tracks the correct memory address for each transformer block and advances through the preallocated buffer at each diffusion iteration, as illustrated by the blue box in Fig. 6b. This approach eliminates unnecessary copy-and-allocation operations in the baseline dynamic KV cache management scheme. The Hugging Face Transformers library provides a similar option through the `StaticCache` option. However, our contribution lies in identifying an appropriate KV-cache management scheme for Alpamayo.

C. Optimization 2: Removing GPU Kernel Launch Overhead

Through our static KV cache management, both the memory address and size of the KV cache are fixed across diffusion iterations. Since the same GPU kernels are executed at every iteration, each kernel references identical memory addresses across iterations, as every iteration is identical in terms of input and tensor dimensions.

This deterministic pattern of GPU kernel execution and memory access enables an additional optimization: recording and replaying GPU kernel launches for diffusion to reduce diffusion latency, leveraging CUDA graphs as illustrated in Fig. 7. As shown on the left-hand side, a CPU thread launches D GPU kernels sequentially. Each launch incurs kernel launch overhead (denoted as $L_1, L_2, \ldots, L_D$), corresponding to the time required to issue kernel execution requests to the GPU. The dotted lines represent the delay between kernel launch and actual GPU execution, which often results in GPU idle periods. These gaps become more pronounced when the kernel execution time is shorter than the launch overhead, leading to significant underutilization of GPU resources. However, as shown on the right-hand side of Fig. 7, we capture the entire launch sequence and replay it as a single L_{graph} using CUDA graphs. This eliminates per-kernel launch overhead and the associated idle gaps, thereby improving overall GPU utilization.

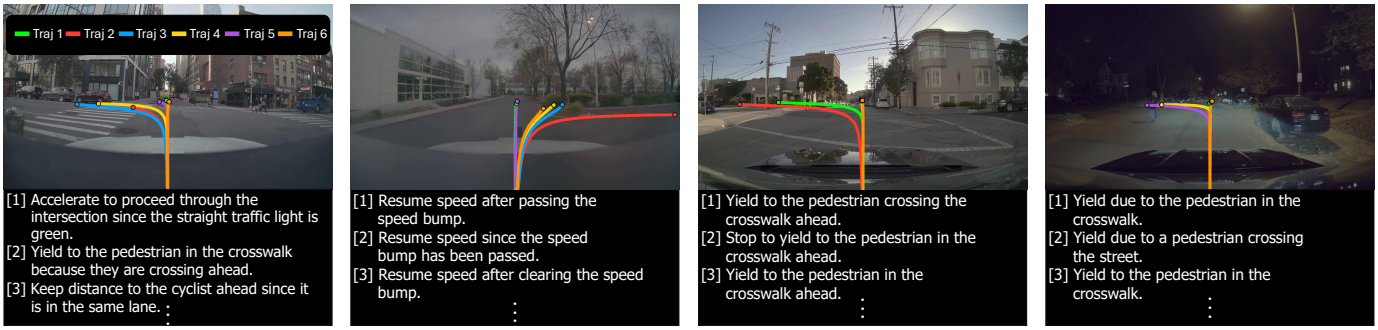
V. Experiments

A. Implementation

Baseline. Our implementation builds upon the Alpamayo inference codebase. We do not modify model parameters, as our contributions are confined to the inference architecture, specifically targeting batch processing and KV cache management. We only modify the Alpamayo inference code to apply our optimizations, while modifying the Hugging Face Transformers library solely for profiling purposes. We also note that Alpamayo 1.5 [33] was recently released, extending Alpamayo with navigation-conditioned reasoning, post-reinforcement learning, and other features unrelated to inference performance. In our preliminary evaluation, Alpamayo 1.5 demonstrates improved trajectory prediction compared to the original Alpamayo. However, since both models share the same inference architecture, our optimization techniques can be directly applied to Alpamayo 1.5 without modification.

Single-reasoning architecture. The official Alpamayo inference code provides a hyperparameter named `num_traj_samples` to control the number of generated trajectories. However, this approach replicates the input to the reasoning module along the batch dimension for each trajectory, resulting in redundant reasoning generation. We avoid this by setting `num_traj_samples` to 1. Instead, after reasoning generation completes, the resulting KV cache is duplicated along the batch dimension to match the desired number of trajectories. The expanded KV cache is then passed to the trajectory generation module to generate multiple trajectories. This modification enables efficient single-reasoning, multi-trajectory generation while preserving the model parameters.

Optimized action generation. We apply two optimizations to the action generation module, targeting KV cache management and GPU kernel execution. First, we replace dynamic KV cache management with our static scheme. The original `torch.cat`-based KV cache updates are eliminated; instead, memory is preallocated prior to the diffusion process, and the cache is updated via index-based in-place assignment. This enables the reuse of memory space for KV cache without unnecessary memory allocations. Second, we incorporate CUDA graphs to capture and replay kernel launches across diffusion iterations. Graph capture is performed during the second iteration, after which subsequent iterations are executed



(a) Multi-reasoning results

(b) Single-reasoning results

Fig. 8: Trajectory diversity comparison between multi-reasoning and single-reasoning with six trajectories with CoT messages

TABLE I: Trajectory prediction performance comparison between multi-reasoning and single-reasoning

Architecture	Open-Loop	Closed-Loop (AlpaSim)
	minADE$_6$@6.4s	DTF (m)
Multi-reasoning	0.725	131.58
Single-reasoning	0.743	137.12

via CUDA graph replay, significantly reducing the kernel launch overhead.

B. Evaluation Results

Trajectory diversity. The latency analysis comparing the single-reasoning and multi-reasoning architectures has already been presented in Section III. In addition, Fig. 8 provides a visual comparison of trajectory diversity using four representative scenarios from the NVIDIA Physical AI Dataset. Although the two approaches do not produce identical trajectories, the single-reasoning results are visually similar to those of multi-reasoning in terms of trajectory diversity. Similar trends are consistently observed across other scenarios beyond the four shown in Fig. 8. This similarity indicates that randomly initialized action sequences provide sufficient diversity for multi-trajectory generation, whereas generating multiple independent reasonings often produces overlapping results and fails to provide adequate diversity.

Trajectory prediction. Since diversity alone does not guarantee the autonomous driving performance, we compare the open-loop and closed-loop trajectory prediction benchmark results in Table I. First, we conduct an open-loop evaluation to measure prediction accuracy. The evaluation follows the same setup as [7] and uses minADE6@6.4s as the evaluation metric. The dataset consists of 394 scenarios, each with a duration of 20 seconds. For each scenario, we generate six trajectory samples and compute the minimum average displacement error (ADE) among them over a 6.4-second prediction horizon. The results show that our single-reasoning approach achieves a minADE6@6.4s score of 0.743, which is comparable to 0.725 obtained by the multi-reasoning approach, indicating no significant difference in prediction accuracy.

Next, we evaluate driving stability in a closed-loop setting using the AlpaSim simulator. NVIDIA used a metric called "AlpaSim score" to report driving stability [7]. However, the detailed definition and implementation of this metric are not publicly available. We therefore introduce a new metric called Distance-to-Failure (DTF), which is the distance traveled by the ego vehicle before the first failure event occurs. A failure event is defined as one of the following: (i) a collision with surrounding traffic participants, (ii) leaving the drivable area, and (iii) a lateral deviation exceeding 4 m from the ground-truth trajectory. This metric reflects the robustness of generated trajectories in interactive driving scenarios. The results show that our single-reasoning approach achieves 137.12 m, which is even slightly longer than the 131.58 m achieved by the multi-reasoning approach. This result indicates that the proposed single-reasoning approach does not degrade closed-loop driving stability.

Why efficient action generation is important. Fig. 9 shows the proportion of action generation latency within the overall inference latency as the number of trajectories increases in both multi- and single-reasoning architectures. When generating a single trajectory, the contribution of action generation is nearly identical between the two architectures, accounting for 18.13 % in multi-reasoning and 18.41 %

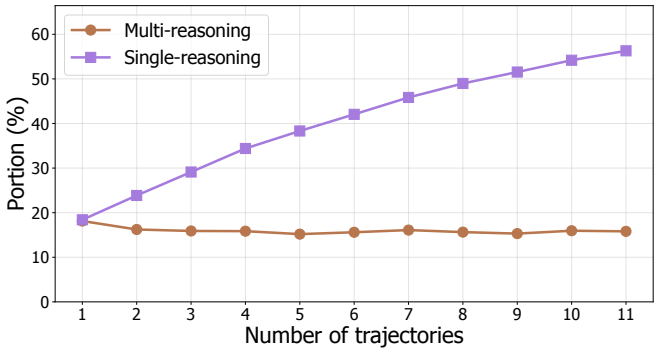


Fig. 9: Proportion of action generation latency with varying numbers of trajectories

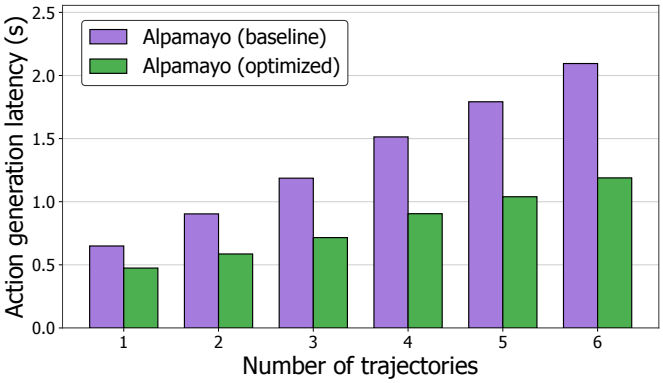
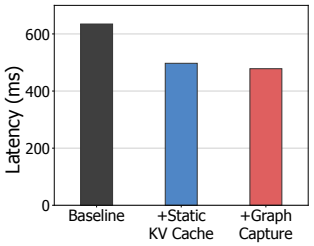
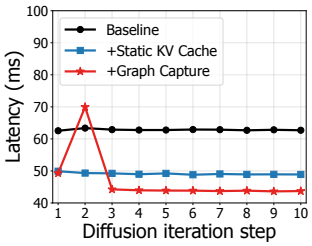


Fig. 11: Action generation latency optimization result



(a) Latency comparison by applying optimization techniques



(b) Per-iteration comparison

Fig. 10: Action generation latency optimization results

in single-reasoning, respectively. However, as the number of trajectories increases, the contribution grows significantly in the single-reasoning architecture, while it remains stable in the multi-reasoning architecture. For instance, with six trajectories, the action generation proportion is just 15.61 % in multi-reasoning, but it reaches as high as 42.06 % in single-reasoning. When the number of trajectories is eleven, the contribution remains around 15.82 % without a visible increase in multi-reasoning, whereas it reaches 52.30 % in single-reasoning, accounting for more than half of the total inference latency. These results indicate that, in the single-reasoning architecture, the action generation latency can be the dominant latency component as the number of trajectories increases, highlighting the importance of our action generation optimization.

Action generation optimization results. Fig. 10a shows the latency reduction achieved by our two optimization techniques within a single trajectory generation. By removing unnecessary copy and allocation overhead (denoted as +Static KV Cache), we achieve a 21.66 % latency reduction compared to the baseline. Furthermore, by additionally applying the CUDA graph capture (denoted as +Graph Capture), the remaining GPU kernel launch overhead is eliminated, yielding a total reduction of 24.65 %. To provide a more fine-grained analysis, Fig. 10b shows the latency breakdown across the 10 diffusion iterations. Overall, the per-iteration latency is reduced from 62.83 ms to 49.13 ms with static KV cache management, and further to 47.00 ms with CUDA graph capture. Note that CUDA graph capture requires a preceding warmup iteration to complete memory allocations. Thus, graph capture is performed at the second iteration, resulting in an elevated latency of 69.95 ms as shown in the figure. However, from the third iteration onward, the captured graph is replayed without GPU kernel launch overhead, yielding an average per-iteration latency of 44.83 ms. Thus, despite the recording overhead at the second iteration, CUDA graph capture reduces overall trajectory generation latency. If more diffusion iterations are required for higher-precision trajectory generation, the benefit becomes even more significant.

We next analyze how our optimization affects the action generation latency with varying numbers of trajectories. Fig. 11 shows that our optimization consistently reduces the action generation latency. One interesting observation is that the relative reduction continually grows as the number of trajectories increases. The reduction proportion is just 27.69 % (from 0.65 s to 0.47 s) when the number of trajectories is 1. However, it continually increases up to 43.06 % (from 2.09 s to 1.19 s) at the six-trajectory case. If the trajectory generations are sequential, the latency reduction proportion itself should remain constant, regardless of the number of trajectories. However, Alpamayo performs multi-trajectory diffusion in a batched manner, which significantly increases the memory bandwidth demand as the number of trajectories grows. This leads to a memory bandwidth bottleneck. In such scenarios, our static KV cache management not only reduces latency but also substantially alleviates the memory bandwidth burden. Consequently, the impact of our optimization is further amplified when dealing with a large number of trajectories.

Overall optimization result. Fig. 12 shows the overall latency comparison between the baseline Alpamayo and our optimized version incorporating both the single-reasoning architecture and the action generation optimization. Our approach consistently reduces latency across all trajectory counts, with the improvement becoming increasingly significant as the number of trajectories grows. In particular, at six trajectories, we achieve up to a 69.23 % reduction in overall latency (from 13.33 s to 4.10 s) compared to the baseline, demonstrating

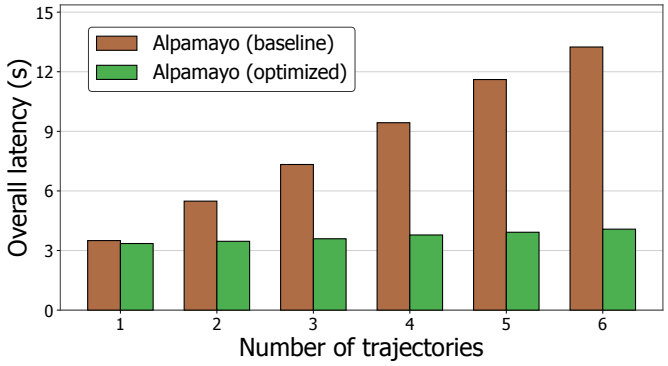


Fig. 12: Overall latency optimization result

the effectiveness of combining single-reasoning with efficient action generation. This reduction is critical, as the baseline latency poses a significant barrier to practical deployment, whereas our optimized latency brings VLA-based autonomous driving models closer to real-world applicability.

VI. Related Work

In this section, we situate our work within reasoning-based E2E autonomous driving systems. Conventional E2E autonomous driving systems lack transparency in their decision-making processes, motivating recent approaches to incorporate VLMs and VLAs to generate human-readable reasoning that enables interpretability of driving decisions [7]–[21]. Existing approaches can be classified into two categories based on how reasoning relates to trajectory generation: (1) multi-reasoning, where a separate reasoning sequence is generated for each trajectory, and (2) single-reasoning, where a single reasoning sequence is shared across all trajectories.

A. Multi-Reasoning E2E Autonomous Driving

The majority of reasoning-based E2E autonomous driving systems adopt the multi-reasoning paradigm [7]–[17]. This paradigm is motivated by the intuition that diverse reasoning traces lead to diverse behaviors. However, since all reasoning sequences are conditioned on the same inputs, their variability is primarily driven by stochasticity in token sampling during reasoning generation rather than systematically different scene interpretations. Consequently, increased reasoning diversity may not consistently translate into meaningful behavioral diversity, as we empirically demonstrate in Section V. Moreover, each trajectory sample requires a separate reasoning pass, resulting in substantial inference overhead as the number of samples grows.

B. Single-Reasoning E2E Autonomous Driving

The limitations of multi-reasoning paradigm motivate the adoption of the single-reasoning paradigm [18]–[21]. In this approach, a single reasoning representation is computed once and shared across all trajectories, with trajectory diversity introduced through stochasticity during trajectory generation. This design reduces inference overhead by eliminating the need to generate multiple reasoning sequences.

However, existing approaches adopt this paradigm in a form that reduces interpretability. ORION [18] generates human readable reasoning, but compresses it into a single planning token that conditions generation of trajectories. DiffVLA [19] conditions its trajectories with VLM-generated driving commands alongside anchor trajectory embeddings. IRL-VLA [20] encodes semantic commands from a VLM module and geometric features from a BEV encoder into a unified conditioning signal for a diffusion planner. ColaVLA [21] replaces explicit reasoning with latent meta-action representations that guide its planner. In these approaches, the representation that directly conditions trajectory generation is a compressed surrogate derived from reasoning rather than the full internal representation of it. As a result, where human-readable reasoning exists, it cannot be treated as a reliable explanation of the generated trajectories, limiting its utility for safety validation.

In contrast, we apply the single-reasoning paradigm to Alpamayo without reducing interpretability. We preserve the CoT reasoning of Alpamayo to directly condition trajectory generation, and further reduce inference latency by identifying and eliminating architectural inefficiencies in the diffusion-based trajectory generation process. As a result, our method achieves lower inference latency than the baseline without sacrificing trajectory diversity, prediction quality, or interpretability.

VII. Conclusion

In this paper, we present a comprehensive latency analysis and optimization of Alpamayo, a reasoning-based E2E autonomous driving system. Through detailed profiling of its inference process, we identified two dominant bottlenecks: the multi-reasoning architecture, which causes inference latency to scale linearly with the number of trajectories, and the diffusion-based action generation, which incurs substantial inter-iteration overhead from unnecessary memory copy operations and inefficient GPU kernel launches. To address these bottlenecks, we proposed two targeted optimizations. First, we redesigned Alpamayo into a single-reasoning architecture that generates a shared reasoning once and reuses it across all trajectories, eliminating redundant computation. Second, we eliminated inter-iteration overhead in diffusion-based action generation through memory pre-allocation and CUDA graph capture. Evaluated through both open-loop and closed-loop experiments, the combined optimizations achieve up to a 69.23 % reduction in end-to-end inference latency while maintaining trajectory diversity and prediction quality. These results demonstrate that jointly analyzing system architecture and runtime execution is essential for improving the practical efficiency of reasoning-based E2E autonomous driving systems. We hope this work will inspire future studies on VLA-based autonomous driving within the real-time systems community.

ACKNOWLEDGMENT

This work was supported by Samsung Electronics Co., Ltd. (IO251217-14751-01). J.-C. Kim is the corresponding author.

REFERENCES

[1] F. Codevilla *et al.*, "End-to-end driving via conditional imitation learning," in *Proceedings of the IEEE International Conference on Robotics and Automation*, 2018.

[2] B. Jiang *et al.*, "Vad: Vectorized scene representation for efficient autonomous driving," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2023.

[3] K. Chitta *et al.*, "Transfuser: Imitation with transformer-based sensor fusion for autonomous driving," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2023.

[4] X. Jia *et al.*, "Drivetransformer: Unified transformer for scalable end-to-end autonomous driving," in *Proceedings of the International Conference on Learning Representations*, 2025.

[5] Y. Hu *et al.*, "Planning-oriented autonomous driving," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2023.

[6] B. Liao *et al.*, "Diffusiondrive: Truncated diffusion model for end-to-end autonomous driving," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2025.

[7] Y. Wang *et al.*, "Alpamayo-R1: Bridging reasoning and action prediction for generalizable autonomous driving in the long tail," arXiv preprint, 2026. [Online], Available: https://arxiv.org/abs/2511.00088 [Accessed: March 25, 2026].

[8] J.-J. Hwang *et al.*, "EMMA: End-to-end multimodal model for autonomous driving," *Transactions on Machine Learning Research*, 2025.

[9] Z. Xu *et al.*, "DriveGPT4: Interpretable end-to-end autonomous driving via large language model," *IEEE Robotics and Automation Letters*, 2024.

[10] J. Mao *et al.*, "GPT-Driver: Learning to drive with gpt," in *NeurIPS Workshop*, 2023.

[11] X. Tian *et al.*, "DriveVLM: The convergence of autonomous driving and large vision-language models," in *Proceedings of the Conference on Robot Learning*, 2024.

[12] B. Jiang *et al.*, "Senna: Bridging large vision-language models and end-to-end autonomous driving," arXiv preprint, 2024. [Online], Available: https://arxiv.org/abs/2410.22313 [Accessed: March 25, 2026].

[13] Z. Zhou *et al.*, "AutoVLA: A vision-language-action model for end-to-end autonomous driving with adaptive reasoning and reinforcement fine-tuning," in *Proceedings of the Advances in Neural Information Processing Systems*, 2025.

[14] Z. Yuan *et al.*, "AutoDrive-R^2: Incentivizing reasoning and self-reflection capacity for vla model in autonomous driving," in *Proceedings of the International Conference on Learning Representations*, 2026.

[15] Y. Luo *et al.*, "AdaThinkDrive: Adaptive thinking via reinforcement learning for autonomous driving," arXiv preprint, 2025. [Online], Available: https://arxiv.org/abs/2509.13769 [Accessed: March 25, 2026].

[16] S. Zeng *et al.*, "FutureSightDrive: Thinking visually with spatio-temporal cot for autonomous driving," in *Proceedings of the Advances in Neural Information Processing Systems*, 2025.

[17] X. Liu *et al.*, "ReasonPlan: Unified scene prediction and decision reasoning for closed-loop autonomous driving," in *Proceedings of the Conference on Robot Learning*, 2025.

[18] H. Fu *et al.*, "ORION: A holistic end-to-end autonomous driving framework by vision-language instructed action generation," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2025.

[19] A. Jiang *et al.*, "DiffVLA: Vision-language guided diffusion planning for autonomous driving," arXiv preprint, 2025. [Online], Available: https://arxiv.org/abs/2505.19381 [Accessed: March 25, 2026].

[20] A. Jiang *et al.*, "IRL-VLA: Training an vision-language-action policy via reward world model," arXiv preprint, 2025. [Online], Available: https://arxiv.org/abs/2508.06571 [Accessed: March 25, 2026].

[21] Q. Peng *et al.*, "ColaVLA: Leveraging cognitive latent reasoning for hierarchical parallel trajectory planning in autonomous driving," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2026.

[22] NVIDIA, "PhysicalAI autonomous vehicles dataset," [Online], Available: https://huggingface.co/datasets/nvidia/PhysicalAI-Autonomous-Vehicles [Accessed: March 25, 2026].

[23] NVlabs, "AlpaSim: A modular, lightweight, and data-driven research simulator for autonomous driving," 2025. [Online]. Available: https://github.com/NVlabs/alpasim. [Accessed: Mar. 25, 2026].

[24] NVlabs, "Alpamayo 1," 2025. [Online]. Available: https://github.com/NVlabs/alpamayo. [Accessed: Mar. 25, 2026].

[25] Paszke *et al.*, "PyTorch: An imperative style, high-performance deep learning library," in *Proceedings of the Advances in Neural Information Processing Systems*, 2019.

[26] Wolf *et al.*, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 2020, pp. 38–45. [Online]. Available: https://arxiv.org/abs/1910.03771

[27] S. Bai *et al.*, "Qwen3-VL technical report," arXiv preprint, 2025. [Online], Available: https://arxiv.org/abs/2511.21631 [Accessed: March 25, 2026].

[28] A. Azzolini *et al.*, "Cosmos-Reason1: From physical common sense to embodied reasoning," arXiv preprint, 2025. [Online], Available: https://arxiv.org/abs/2503.15558 [Accessed: March 25, 2026].

[29] A. Yang *et al.*, "Qwen2.5 technical report," arXiv preprint, 2025. [Online], Available: https://arxiv.org/abs/2412.15115 [Accessed: March 25, 2026].

[30] L. Meng *et al.*, "Deepstack: Deeply stacking visual tokens is surprisingly simple and effective for lmms," in *NeurIPS*, 2024.

[31] N. Muennighoff *et al.*, "s1: Simple test-time scaling," arXiv preprint, 2025. [Online], Available: https://arxiv.org/abs/2501.19393 [Accessed: March 25, 2026].

[32] C. Snell *et al.*, "Scaling llm test-time compute optimally can be more effective than scaling model parameters," arXiv preprint, 2024. [Online], Available: https://arxiv.org/abs/2408.03314 [Accessed: March 25, 2026].

[33] NVlabs, "Alpamayo 1.5," 2026. [Online]. Available: https://github.com/NVlabs/alpamayo1.5. [Accessed: Mar. 25, 2026].