vla.cpp: A Unified Inference Runtime for Vision-Language-Action Models

Supplementary Appendix

About this document

This is the supplementary document to *vla.cpp: A Unified Inference Runtime for Vision-Language-Action Models*. Refer to the README in the source code for downloading the model bundles. Demo videos and the reproducible benchmark scaffold are available at the project website: `https://fai-modelopt-tech.github.io/vla-cpp.github.io/`.

A From Architecture to Implementation

A.1 The served architectures

Table 1 summarizes the seven architectures `vla.cpp` serves. They span four vision-encoder and six language-model architectures across the seven models, yet differ in the action head along exactly the two-form split above: six iterative heads, one single-pass head. BitVLA's single-pass regression head has no solver loop. The GR00T N1.5 and N1.6 vision backbones are the SigLIP2-400M encoders inside the Eagle-2.5 and Eagle3-VL VLM, respectively. GR00T-N1.7 uses the native ViT of the Cosmos-Reason2.

Table 1: The seven served architectures. FM = flow-matching; AltVL = AlternateVL; MLP = multi-layer perceptron; vl-self-attn = vision-language self-attention. T is the number of solver steps the iterative action head integrates per action chunk. The single-pass BitVLA head has no solver loop.

Model	Vision backbone	Language backbone	Action head	Params	T
SmolVLA	SigLIP-So400m	SmolLM2-360M	FM cross-attn expert	450M	10
Evo-1	InternViT-300M	Qwen2.5-0.5B	cross-attn DiT	770M	32
BitVLA	BitSigLIP-L	BitNet-2B	MLP regression	2.4B	-
π_0	SigLIP-So400m	Gemma-2B	FM joint-attn expert	3B	10
GR00T-N1.5	SigLIP2-400M	Qwen3-1.7B	AltVL DiT+vl-self-attn	3B	4
GR00T-N1.6	SigLIP2-400M	Qwen3-1.7B	AltVL DiT	3B	4
GR00T-N1.7	Qwen3-VL ViT	Qwen3-VL	AltVL DiT+vl-self-attn	3B	4

A.2 Implementation of the Prefix Extension

The runtime targets the two-stage structure of Section 3.1 of the main paper: a vision-language backbone encodes a cached multimodal prefix, and a separate action head consumes it to produce an action chunk. The prefix is exposed, masked, and cached inside the `ggml` graph.

1. **Exposing full hidden states.** A text runtime returns only logits or a pooled embedding. We tap the final-layer hidden-state tensor of the language model before the output projection and route the full [tokens $\times$ d] sequence to the action head as the cross-attention source.
2. **Bidirectional prefix mask.** The prefix is encoded with a bidirectional attention mask rather than the causal mask used for decoding, so images, instruction, and state attend freely. We construct the mask at graph-build time from the segment layout of each request.
3. **Cross-attention cache lifecycle.** The prefix keys and values are computed once per observation and reused across all solver steps. We allocate a dedicated cross-attention cache, distinct from the language-model self-attention KV cache, that persists for the lifetime of one denoising integration and is released when the action chunk is returned.

The action head comes in two forms, both served behind the same prefix interface. Most families use an iterative head that cross-attends to the prefix and integrates a chunk over several solver steps: the flow-matching experts of π_0 and SmolVLA, the cross-attention flow head of Evo-1, and the diffusion-transformer head of the GR00T series. BitVLA uses the other form, a single-pass regression head that emits the chunk in one forward pass. The cache lifecycle thus amortizes prefix reuse across solver steps for iterative heads and degenerates to a single read for the single-pass head, with no change to the prefix path. This is why BitVLA recurs throughout the evaluation: its contribution is the first end-to-end ternary stack in a `ggml`-class engine and the custom tensor-core ternary GEMM of Section 4.5 of the main paper, exercising the footprint and kernel axes rather than the iterative-head axis of the other six.

B Per-Task Agreement Across LIBERO Suites

Table 2 reports success rate with 95% Wilson confidence intervals on the `libero_spatial`, `libero_goal`, and `libero_10` suites alongside `libero_object`, for the architecture-wise experiments. Consistent with the reference-matching statement of Section 4.2 of the main paper, the relevant quantity is the gap to the reference, not the absolute rate. Saturated suites bound run-to-run variance and make a one-episode gap meaningful. On all but one suite the runtime and reference confidence intervals overlap, so the two are statistically indistinguishable. The single exception is GR00T-N1.7 on Object task, where both policies are near ceiling (runtime 97.0%, reference 99.8%) and the intervals are narrow enough that a residual 2.8-point gap separates them. The gap is small relative to the reference's own run-to-run spread on the harder suites and does not appear on Spatial, Goal, or Long, so we read it as a near-saturation difference of a few episodes rather than a systematic fidelity loss.

Table 2: Experimental results across the four LIBERO suites (SR in % with 95% Wilson intervals in brackets). `vla.cpp` is our runtime and *ref* is the reference checkpoint reproduced under consistent configuration.

	BitVLA		GR00T-N1.7	
Suite	`vla.cpp`	ref	`vla.cpp`	ref
Spatial	94.5 [90.4, 96.9]	93.6 [91.1, 95.4]	96.0 [92.3, 98.0]	97.3 [95.7, 98.4]
Object	100.0 [98.1, 100.0]	99.6 [98.6, 99.9]	97.0 [93.6, 98.6]	99.8 [99.1, 100.0]
Goal	92.5 [88.0, 95.4]	91.6 [88.8, 93.7]	97.5 [94.3, 98.9]	98.0 [96.5, 98.9]
Long	83.5 [77.7, 88.0]	85.8 [82.5, 88.6]	89.0 [83.9, 92.6]	91.7 [89.2, 93.6]

C Optimized and Cross-Stack Baselines

Section 4.3 of the main paper compares `vla.cpp` against the released eager-mode PyTorch reference. Table 3 adds (i) additional architectures and (ii) a graph-captured PyTorch configuration (`torch.compile` with a static cache), isolating the share of the speedup attributable to the runtime rather than to eager-mode kernel dispatch. This configuration is reported only where the full forward path lowers into a compiled region; for the architectures whose backbone cannot be captured (discussed at the end of this section) the entry is left empty. We were unable to build an end-to-end engine for the iterative cross-attention action head with the single-vendor compiler stacks on the Orin parts. The capability matrix in Table 1 of the main paper records why each stack does not provide an apples-to-apples baseline for this inference pattern.

The three architectures fall into two regimes: the two compact backbones where the runtime delivers a clear compute-only speedup, and the large backbone where it matches eager mode. The forward-computation latencies are reported at the server side, excluding the small ZMQ transport overhead that the deployed runtime additionally incurs.

For SmolVLA the runtime delivers a $2.4\times$ compute-only speedup over eager PyTorch and edges out the graph-captured configuration (74.1 vs 76.3 ms). This confirms that most of the gain is structural rather than an artifact of eager kernel dispatch. GR00T-N1.7 is the more demanding case and matches eager mode's result: `vla.cpp` lands at 101.4 ms, statistically level with eager PyTorch (101.7 ms).

Table 3: Per-step latency (ms): eager PyTorch, graph-captured PyTorch (`torch.compile` with a static cache), and `vla.cpp`. The RTX 5070 rows are the median over 100 timed steps; the GR00T-N1.6 row is the per-step latency of the on-robot ALOHA run (per-chunk inference over an eight-step chunk in Table 4 of the main paper). A dash (−) marks an architecture for which no fully graph-captured configuration is available.

Model	Device	PyTorch (eager)	PyTorch (graph-captured)	`vla.cpp`
SmolVLA	RTX 5070	175.99	76.26	**74.11**
GR00T-N1.7	RTX 5070	101.67	-	**101.38**
GR00T-N1.6	AGX Orin	77.5	-	**58.8**

On the embedded AGX Orin, the GR00T-N1.6 row reports the per-step latency of the on-robot ALOHA run. In this case, `vla.cpp` reaches 58.8 ms against 77.5 ms for eager PyTorch (470 ms vs 620 ms per eight-step chunk), the gap that drives the closed-loop success difference analyzed in Table 4 of the main paper.

For GR00T-N1.7 the graph-captured baseline is left empty for a structural reason rather than because of insufficient tuning. Its Qwen3-VL backbone routes self-attention through an external FlashAttention CUDA extension rather than through native, traceable ATen operators. Two properties of that path defeat both capture mechanisms. First, the kernel is opaque to `torch.compile`: TorchDynamo cannot trace into the external operator and inserts a graph break at its boundary, so the backbone never lowers into a single compiled region and falls back to eager dispatch. Second, the variable-length attention path computes its cumulative-sequence-length metadata (`cu_seqlens`) on the host and launches kernels over data-dependent, dynamic shapes with host-to-device synchronization. These operations are not permitted inside a CUDA-graph capture region, which requires fully static shapes and no host synchronization. Only the action head satisfies these constraints, as it consists of dense, static-shape attention and GEMMs over a fixed horizon and four denoiser steps. Capturing it in isolation, however, leaves the dominant backbone in eager mode and does not yield a meaningfully different end-to-end latency. Both stacks are therefore bottlenecked by the same large backbone, and the runtime's broader advantage for this architecture remains its deployment footprint, portability across the Orin tiers, and the ternary and flow-expert paths it uniquely supports.

D Per-Component Roofline Breakdown

Table 4 decomposes the single-request forward path of π_0, a representative large-backbone deployment, into two phases: the compute-bound prefix, which combines the vision encoder and the language-model prefill, and the memory-bound action expert. Both stages of the prefix are dense many-token passes with high weight reuse, which places them in the compute-bound regime. For each phase we report the measured latency share, the operational intensity, and the regime relative to the device balance point. This makes explicit the two-phase structure discussed in Section 4.5 of the main paper.

We instrument three architectures that span the backbone-size range, SmolVLA, GR00T-N1.6, and π_0, and report their per-phase split in Table 5. The prefix phase accounts for the majority of the forward across all three models. It does so decisively for the large-backbone π_0 ($\sim$75%), and by a narrower margin for the compact SmolVLA and GR00T-N1.6 ($\sim$52% each). For the two compact models the memory-bound action expert grows to a near-equal share ($\sim$48%), because its cost rises with the solver-step count and, for GR00T-N1.6, with a 32-layer cross-attention DiT. Because the device balance point (the ridge of the roofline) differs across hardware, both the per-phase latency split and, near the balance point, the regime assignment can shift between tiers. However, the operational intensity of each phase is a property of the computation itself and is device-independent. It is this quantity that places the dense prefix far above and the action expert far below the balance point on the devices we measure.

For π_0 the prefill and the per-step denoise are separated by varying T and linear-fitting the fused inference graph (inference(T) = prefill + $T \cdot$ per-step). For SmolVLA the runtime reports the two phases directly. The compute-bound prefix is the larger phase for every model. It dominates decisively for the large-backbone π_0 and for the single-pass BitVLA. For the compact SmolVLA (ten solver

Table 4: Per-phase latency share and operational intensity (FLOPs/byte) for π_0 (PaliGemma-3B backbone + flow action expert, 10 solver steps), a representative large-backbone deployment, on the RTX 5070. The compute-bound prefix dominates the forward, while the memory-bound action expert's share grows linearly with the solver-step count (≈ 5.4 ms/step measured).

Phase	Latency share (%)	Op. intensity (FLOPs/B)	Regime
Prefix (vision + LM)	75	$\sim$256-530	compute-bound
Action expert	25	$\sim$50	memory-bound
Total forward path	100	-	prefix-dominated

steps) and GR00T-N1.6, the margin is narrow, and the memory-bound action expert approaches an equal share. GR00T-N1.6 reaches this near-equal split even at only four solver steps because of its heavy 32-layer cross-attention DiT. Both the shares and the regime assignment are device-dependent near the balance point (Sec. 4.5 of the main paper); each phase's operational intensity is not.

Table 5: Per-phase latency share (% of the forward compute) across four architectures, measured on the RTX 5070 (median over timed iterations, two camera views). T is the number of solver steps the iterative action head integrates per chunk (Table 1); the single-pass BitVLA head is marked $-$. The prefix phase combines the vision encoder and the language-model prefill.

Model	Language backbone	Size	T	Prefix	Action expert
SmolVLA	SmolLM2-360M	450M	10	**52**	48
GR00T-N1.6	Qwen3-1.7B	3B	4	**52**	48
π_0	Gemma-2B	3B	10	**75**	25
BitVLA	BitNet-2B	2.4B	-	**99.5**	0.5

BitVLA is the extreme of this decomposition. As mentioned in Appendix A, its single-pass action head has no solver loop. The prefix phase, formed by its one-shot 30-layer BitNet prefill together with the vision encoder, is therefore $\sim$99.5% of the forward, and the action head $\sim$0.5% (RTX 5070, measured). This is the cleanest instance of the prefix-compute-bound regime: nearly the entire forward is the high-arithmetic-intensity prefix that the kernel optimization targets.

E Reduced-Precision Ablation

This appendix expands the reduced-precision finding of Section 4.6 of the main paper. The vision encoder selects a row of a position-embedding table using an index computed from the patch grid, and the rounding of this index differs between 32-bit and 16-bit floating point: a value just below one rounds down in single precision but up in half precision. For SmolVLA, the model is deployed in half precision while the position index is still computed in single precision. This mismatch selects a different embedding row for almost every patch, which shifts the encoded prefix. The shift is large enough to displace the predicted action chunk by up to approximately 1.97 in the policy's denormalized action units (Table 6). These units are the 7-DoF `libero_object` command, a delta end-effector pose plus a gripper signal recovered by the policy's mean/std unnormalization, and they are dimensionless rather than metric. With per-degree-of-freedom action standard deviations of 0.34-0.44 (translation), 0.04-0.08 (rotation), and $\approx$1.0 (gripper), a displacement of 1.97 is a near-full-scale ($\sim 2\sigma$) error concentrated on the gripper channel. The command saturates to the opposite extreme rather than drifting, which is why the grasp fails outright. This displacement is the difference between grasping the target object and reaching past it: the `libero_object` task never completed, yet no component of the pipeline reported an error.

Making the index precision-aware restored numerical agreement, dropping the position-embedding relative error from 0.32 to 0.003 and recovering the task. The mechanism of reduced-precision collapse and its downstream behavioral effects are themselves documented for language and vision encoders [1, 2]. Our contribution is to quantify it in robot action space and to gate against it. To separate a property of reduced precision in general from a property of the specific operation, Table 7 contrasts SmolVLA with GR00T-N1.6. GR00T-N1.6's vision encoder carries an analogous position-embedding step that is continuous rather than discrete. When its input resolution differs from the native grid, it resizes the position-embedding table by bilinear interpolation instead of selecting a

row by a computed index. For each model we report `libero_object` success at the mismatched versus precision-aware configuration, together with the growth of the action-chunk displacement with solver-step count.

Table 6: Effect of the SmolVLA vision-encoder position-index precision on the encoded prefix and on `libero_object` task completion. A single rounding difference, invisible to an error-free run, moves the action chunk far enough to miss the object.

Position-index precision	pos.-embed rel. err.	action chunk $\max \lvert\Delta\rvert$	task completes
Mismatched (half model, single index)	0.32	1.97	no
Precision-aware (matched)	0.003	$< 10^{-2}$	yes

Table 7: Precision-causality ablation on `libero_object` (10 episodes/configuration, task 0). The catastrophic failure is specific to the discrete position-index selection: SmolVLA's success collapses and its action displacement compounds with solver steps, whereas GR00T-N1.6's continuous interpolation perturbs the position table by only $\sim 0.1\%$ and is behaviorally inert.

Model	Vision position op	SR (mismatched)	SR (aware)	action $\max \lvert\Delta\rvert$ vs. steps
SmolVLA	discrete index	20.0	90.0	$0.26 \rightarrow 1.05$ (grows)
GR00T-N1.6	continuous interp.	100.0	100.0	~ 0.003 (flat)

The comparison localizes the failure to the kind of operation, not to reduced precision as such. SmolVLA's position index is a discrete table lookup, so a single rounding difference selects a wrong row. This is a discontinuous jump in the encoded prefix that the flow-matching solver then amplifies: the mismatched-versus-aware action displacement grows from 0.26 at one Euler step to 1.05 at eight, while task success falls from 90% to 20%. This growth is not solver noise. Across the same range of solver steps, the precision-aware policy's own discretization error decreases toward its refined solution, so the two configurations converge to two distinct fixed points. The perturbation is therefore a systematic bias that the integrator settles into, rather than a transient. GR00T-N1.6's bilinear table resize is by contrast a continuous operation. Evaluating it in half versus single precision moves the position table by only $\sim 0.1\%$ relative, an effect further attenuated by the bf16-resident weights. It leaves the action chunk unchanged across solver steps and does not flip a single episode (100% in both configurations). Catastrophic precision sensitivity therefore requires an operation that rounds across a discrete boundary, whereas continuous operations on the same encoded quantity degrade gracefully. The distinction is directly actionable for a runtime: index, `argmax`, and bucketize selections must be made precision-aware and gated, whereas the surrounding continuous algebra tolerates reduced precision.

F SimplerEnv (WidowX) Results

As a second simulator and embodiment, we evaluate GR00T-N1.6 on the WidowX (bridge) tasks of SimplerEnv at a 252-pixel vision resolution. GR00T-N1.6 reaches 80% on the "put carrot on plate" task and 70% on the "put spoon on towel" task. The two more complex tasks, "stack the green block on the yellow block" and "put eggplant in the yellow basket", yield lower success rates. This pattern matches the reference policy's own difficulty profile, consistent with the parity claim of Section 4.2 of the main paper.

G Replan Interval and the Latency-Staleness Trade-off

The closed-loop argument of Section 4.7 of the main paper rests on a trade-off between inference cost and observation freshness, set by how many steps of a predicted action chunk are executed before the policy re-plans. We make this trade-off explicit by varying the replan interval S, the number of executed steps per inference call, for SmolVLA on the `libero_object` suite, deployed on the Jetson Orin Nano. All other factors are held fixed: a single seed, an identical observation pipeline, fp16 weights with precision-aware index gating, and 50 episodes per setting. Small S re-plans often and acts on fresh observations at high inference cost; large S executes the chunk open-loop and acts on increasingly stale observations.

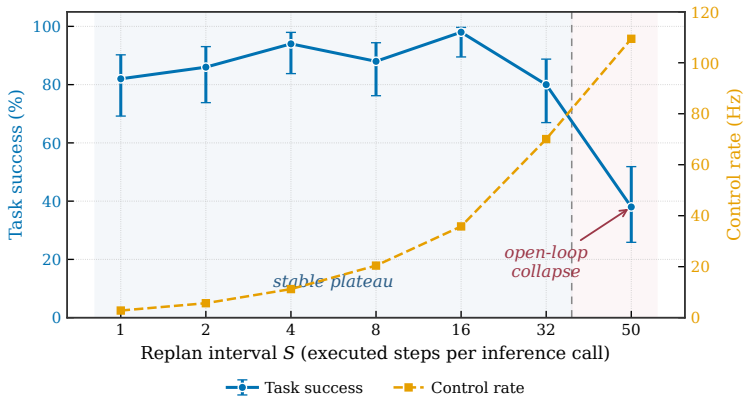

Figure 1: Replan-interval study for SmolVLA on `libero_object` (Jetson Orin Nano, 50 episodes/setting). Task success (left axis, 95% Wilson intervals) is stable across the shaded plateau and collapses once execution becomes fully open-loop at S=50. The amortized control rate (right axis) rises almost linearly with S.

Table 8: Replan-interval study for SmolVLA on `libero_object` (Jetson Orin Nano, 50 episodes/setting). S is the executed steps per inference call; latency is per call, the control rate is amortized over the executed steps, and success carries a 95% Wilson interval.

S	Lat. median (ms)	Lat. p95 (ms)	Control rate (Hz)	Success (%)
1	358.4	370.5	2.8	82.0 [69.2, 90.2]
2	352.7	363.4	5.7	86.0 [73.8, 93.0]
4	356.6	383.2	11.2	94.0 [83.8, 97.9]
8	391.7	429.9	20.4	88.0 [76.2, 94.4]
16	447.0	463.3	35.8	**98.0** [89.5, 99.6]
32	456.7	466.8	70.1	80.0 [67.0, 88.8]
50	457.0	467.9	109.4	38.0 [25.9, 51.8]

Table 8 reports the result. The effective control rate rises almost linearly with S, from 2.8 Hz at S=1 to 109 Hz at S=50, because each inference call amortizes over more executed steps while its own latency stays within a narrow band (353-457 ms). Per-call latency is roughly flat for small S and rises by about a quarter for the longest intervals, consistent with longer uninterrupted execution between cache refreshes. Task success is statistically stable across a broad plateau of intervals from S=1 to S=32, where the per-setting confidence intervals all overlap, and then collapses at S=50 to 38% ($[25.9, 51.8]$), far below every other setting. The collapse coincides with the point where the policy executes its entire chunk without a single intermediate observation, confirming that the failure is driven by observation staleness rather than by inference cost. Figure 1 plots the two axes together and shows the plateau and the open-loop collapse directly.

Two consequences follow for deployment. First, a wide range of replan intervals is safe, so a runtime can trade inference frequency for throughput over the plateau without sacrificing task success, which is the headroom the asynchronous execution of Section 4.7 of the main paper exploits. Second, fully open-loop execution of a long chunk is not safe, which is the same mechanism behind the lower accuracy of the long-horizon π_0 configuration in Table 2 of the main paper: a long action chunk consumed without re-planning drifts from the observation it was conditioned on.

References

[1] J. Yuan, H. Li, X. Ding, others, and Z. Liu. Give Me FP32 or Give Me Death? Challenges and Solutions for Reproducible Reasoning. arXiv:2506.09501, 2025.

[2] P. Qi, Z. Liu, X. Zhou, others, and M. Lin. Defeating the Training-Inference Mismatch via FP16. arXiv:2510.26788, 2025.