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

Khanh D. Nguyen[1], Hung T. Ho[1], Chinh T. Nguyen[1], Thanh Q. Duong[1], Linh D. Le[1], Duy M. H. Nguyen[4,5,6], Vien A. Ngo[1,2], An T. Le[1,2,3,]

[1]VinRobotics, Vietnam [2]Center for AI Research, VinUniversity, Vietnam
[3]Intelligent Autonomous Systems, TU Darmstadt, Germany
[4]Max Planck Research School for Intelligent Systems, Germany
[5]University of Stuttgart, Germany [6]German Research Center for Artificial Intelligence

***Abstract*—Vision-Language-Action (VLA) policies are typically shipped as Python/PyTorch stacks that assume a workstation-class GPU, a mismatch for the hardware on which robots actually run. We present `vla.cpp`, a portable C++ inference runtime built on `llama.cpp`. To our knowledge it is the first `ggml`-class engine to natively serve the flow-matching and diffusion VLA inference pattern, in which a cached vision-language prefix is consumed by a cross-attending action expert integrated over several solver steps. A single runtime serves seven architectures spanning five backbone and four action-head families behind one request/response protocol, with each model packaged as a self-contained bundle. On LIBERO-Object, the engine matches a state-of-the-art checkpoint to within one episode out of 200, and runs BitVLA at 100% success in 1.3 GiB of memory. The same bundle runs unchanged across three hardware tiers, from a consumer GPU down to an 8 GB embedded module. We then deploy the runtime on a physical ALOHA arm for closed-loop pick-and-place, where lower per-chunk latency lets the policy replan against fresher observations and substantially improves task success over the PyTorch reference on the same module. Code is available at https://github.com/VinRobotics/vla.cpp.**

I. Introduction

A vision-language-action (VLA) model is a robot policy that maps camera images and a natural-language instruction directly to robot actions. Most modern open VLAs [19, 20, 16] share a computational structure: a pretrained vision-language model (VLM) encodes the images and instruction into a multimodal *prefix*, and a separate flow-matching or diffusion *action head* cross-attends to that prefix and iteratively denoises an *action chunk* of several future actions, which the robot replays open-loop before querying again. These policies are almost always released as Python/PyTorch stacks that presume a workstation-class GPU, whereas a robot's onboard computer is typically a memory-constrained embedded module, most often a Jetson Orin, whose unified memory starts at 8 GB and is shared with the OS, camera pipeline, and the rest of the autonomy stack. For LLM this shortfall is met by portable inference engines [4, 9] and single-vendor compiler stacks [17, 18] on the latest accelerators, but none supports the VLA inference pattern across many architectures and heterogeneous edge hardware, the gap `vla.cpp` fills.

We present `vla.cpp`, a C++ inference runtime built on `llama.cpp`. It adds the components a VLM runtime omits: a cross-attention key/value cache reused across solver steps, and flow-matching and diffusion action heads. A single binary serves seven VLA architectures that span five vision-language backbones and four action-head families. Each model is packaged as one self-contained bundle file. A C++ server exchanges observations and action chunks with a lightweight client over a compact binary protocol, so the same engine drives either a simulator or a physical robot without modification. The system currently builds for CUDA and CPU, and its backend-agnostic structure accommodates further targets, such as Apple Silicon GPUs. Our contributions includes:

1) **A unified VLA inference runtime.** We present `vla.cpp`, to our knowledge the first `ggml`-class C++ engine to natively implement cross-attention into a cached VLM prefix across iterative denoising steps for flow-matching and diffusion action heads. A single binary and one bundle format support seven architectures across five backbones and four action-head families
2) **A cross-hardware evaluation of seven state-of-the-art VLA models on three deployment tiers.** We evaluate all seven architectures on LIBERO-Object across a consumer RTX 3060, a Jetson AGX Orin, and an Orin Nano. The engine reproduces each reference checkpoint while running it up to $7.95\times$ faster than PyTorch
3) **An end-to-end deployment on a real ALOHA robot.** On two physical manipulation tasks, `vla.cpp` drives GR00T-N1.6 closed-loop against the PyTorch reference on the same module, reaching $35/40$ (87.5%) success versus $16/40$ (40.0%); its ≈ 150 ms lower per-chunk latency keeps each replan against a fresher observation and compounds over long horizons.

II. Related Work

Early generalist VLAs such as OpenVLA [8] treat control as autoregressive generation, discretizing each action dimension into tokens that a vision-language model emits one at a time. A second and now dominant design replaces the token decoder with a continuous *action head* that is conditioned on the multimodal prefix and trained by flow matching or diffusion [1]. This design produces smooth, high-rate action chunks without

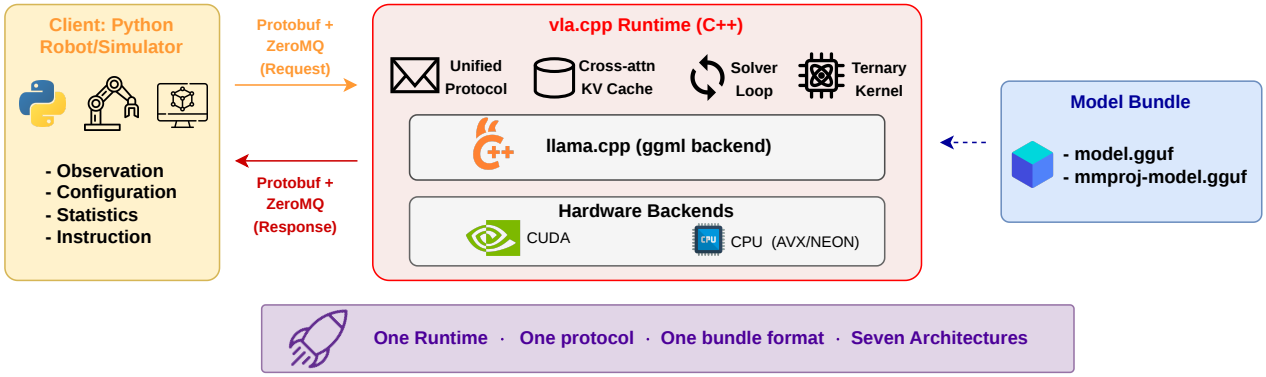

Fig. 1. Architecture of `vla.cpp`. A stateless C++ server loads one GGUF bundle, runs the VLA inference pattern on each Protobuf observation, and returns a denormalized action chunk over ZeroMQ; a thin Python client owns closed-loop control and drives a simulator or robot.

a quantization grid. π_0 [19] and SmolVLA [20] attach a flow-matching expert to a pretrained backbone, GR00T N1.x [16] pairs a VLM with a diffusion-transformer head, and Evo-1 [14] couples an InternVL3 backbone to a cross-attention flow-matching head. BitVLA [22] is, to our knowledge, the first VLA to quantize the backbone to ternary (1.58-bit) weights for an exceptionally small footprint, yet no dedicated inference engine accompanied its release. These models differ in backbones and heads but share the VLA inference pattern (appendix A1), which our runtime targets.

To deploy LLM on edge devices, portable inference engines package a model into one compact file and run it on heterogeneous accelerators or CPUs without a vendor toolchain; `llama.cpp` [4], built on `ggml`, is the most widely deployed, alongside MLC-LLM [15] and vLLM [9]. We build on `llama.cpp` for its portability and open ecosystem, reusing its GGUF bundle format and `mtmd` multimodal extension. On the accelerator side, single-vendor stacks such as TensorRT-LLM [18] and TensorRT Edge-LLM [17] compile one model into one engine for the vendor's newest hardware, offering only preview or per-model support on the widely deployed Orin modules. None of these implements the flow-matching or diffusion action head, the cross-attention into a cached prefix consulted across denoising steps, or the chunked control loop a VLA requires; we therefore treat them as complementary on the portability axis (Table I) and compare against the reference PyTorch implementation from which practitioners start.

III. The `vla.cpp` Engine

A. System architecture

`vla.cpp` runs as two processes (Figure 1). A C++ server loads the model once and holds it resident; on each request it decodes an observation (camera images, language instruction, proprioceptive state) from a Protobuf message, executes the inference pattern, denormalizes the actions into robot units using statistics carried in the bundle, and returns the chunk. The server is deliberately *stateless*: control-history concerns such as blending overlapping chunks, handling stale observations, and safe-holding when inference is late reside on the client, typically Python for interoperability with existing robotics and simulator code. The two sides communicate over ZeroMQ [6] with a Protobuf [5] schema, chosen for low loopback latency, language-agnostic bindings, and a typed contract generated from one schema file. Each model is a single GGUF bundle holding the language model, action head, normalization statistics, and architecture configuration; the vision encoder is embedded or supplied as a companion multimodal file.

B. From VLM to VLA

A VLA decomposes into a vision encoder, a language model, and an action head; the first two form a vision-language model already supported by `llama.cpp` and its multimodal extension, which `vla.cpp` reuses for the prefix. Serving the action head faithfully requires closing four gaps that a text runtime leaves open (appendix A): (i) exposing the language model's full final-layer hidden states, not just logits, as the cross-attention source; (ii) encoding the prefix under a *bidirectional* mask, since the causal mask of a text decoder would silently corrupt every cached key and value; (iii) an explicit cross-attention cache lifecycle that computes the prefix keys and values once and reuses them across all solver steps, which the language-model KV-cache abstraction does not provide; and (iv) denormalizing the predicted chunk with the model's own statistics, a common silent source of incorrect motion. Building on this foundation, `vla.cpp` implements the action heads of all seven architectures, spanning five vision-language backbones and four action-head families

IV. Experiments

A. Experiments in simulation environment

We evaluate all seven architectures on the full `libero_object` suite [11] (ten tasks $\times$ twenty episodes $=$ 200 per architecture), counting any non-completed episode as a failure. We also run GR00T-N1.6 on the WidowX (bridge) tasks of SimplerEnv [10] as a second simulator and embodiment (appendix G). We profile each model on three tiers: a consumer RTX 3060, an embedded Jetson AGX Orin,

TABLE I

WHY COMMON SERVING STACKS ARE COMPLEMENTARY, NOT COMPETING, FOR THE VLA INFERENCE PATTERN ON EDGE. ✓ SUPPORTED; × NOT SUPPORTED; ○ PARTIAL OR PER-MODEL, PER-DEVICE ENGINEERING.

Stack	Portable bundle	Dynamic cross-attn. KV cache	Flow/diffusion action head	Official Orin	Cross-tier, no recompile
TensorRT-LLM [18]	×	○	×	○	×
TensorRT Edge-LLM [17]	×	○	○	○	×
ONNX Runtime / Torch-TensorRT	×	×	×	○	×
MLC-LLM [15]	○	×	×	○	×
`llama.cpp` AR-VLA [25]	✓	×	×	✓	✓
`vla.cpp` (ours)	✓	✓	✓	✓	✓

and a memory-constrained 8 GB Orin Nano, the latter two sharing a unified CPU/GPU memory pool. Our reference throughout is the PyTorch eager implementation that ships with each model.

Beyond reproducing the reference checkpoint's behavior (section A2), `vla.cpp` runs it substantially faster than the released PyTorch reference. Holding model weights, BF16 precision, and the simulator fixed and varying only the runtime, `vla.cpp` runs SmolVLA at 28.16 ms per environment step against 223.96 ms for PyTorch, a $7.95\times$ speedup, at essentially the same peak VRAM (1410 vs 1406 MiB) and with success rates matching within sampling noise (additional architectures and a graph-captured PyTorch configuration in appendix E). The gap follows from how each engine maps the same forward onto the GPU: a single-request SmolVLA forward is hundreds of short CUDA kernels, and under PyTorch eager execution each dispatches from Python, so host-side launch overhead occupies the device and the arithmetic units idle for much of every step. `vla.cpp` inherits from `llama.cpp` a static graph captured once at load time and replayed each forward, paying the per-kernel host cost at capture rather than per call. With the IMMA kernel of section B, this is a second instance in which single-request speed is set by how fully the arithmetic units stay occupied, and the two effects compose.

Carrying the same models down the three tiers (Table II), per-step latency rises smoothly toward the Orin Nano (roughly three to ten times, by architecture) and stays within a real control loop's chunk budget for the lighter models. The decisive effect, however, is capacity: only five of the seven architectures load on the 8 GB Nano. GR00T-N1.6/N1.7 do not fit, as their ≈ 6 GiB of resident weights exhaust the unified pool shared with the OS, camera pipeline, and simulator; GR00T-N1.5 and π_0 fit only with the simulator offloaded to another machine. The physical size of the weights saturates the pool well before computation becomes the bottleneck, so footprint, not latency, determines what can be deployed on an inexpensive module.

B. Real-robot experiments

A real ALOHA arm executes two manipulation tasks of differing horizon: Task 1 picks several blocks into a white box; Task 2 places trash into a box and then a banana on a dish. We compare `vla.cpp` against the released PyTorch reference for GR00T-N1.6, the variant we fine-tuned on ALOHA demonstrations, running the same checkpoint in BF16 on the same module, so any difference reflects the runtime, not the weights. Closed-loop, both engines run the same synchronous loop with an eight-step chunk of which the first seven execute before replanning, so the observation ages by the full inference latency before its first action reaches the arm. Mean per-chunk inference is 470ms for `vla.cpp` versus 620ms for PyTorch; with 840ms of chunk execution in both cases, the replan period is about 1.3 s (0.76 Hz) versus 1.5 s (0.68 Hz).

TABLE II

PER-STEP LATENCY (MS) ACROSS THE THREE TIERS AND RESIDENT FOOTPRINT ON THE 8 GB ORIN NANO. †SIMULATOR OFFLOADED TO A SECOND MACHINE (CO-RESIDENT SERVER, CLIENT, AND SIMULATOR EXCEED 8 GB), SO LATENCY INCLUDES A NETWORK HOP. *DOES NOT FIT: WEIGHTS EXHAUST THE 8 GB POOL.

Model	RTX 3060	AGX Orin	Orin Nano	RSS (MiB)
SmolVLA	28.16	65.41	141.81	2031
BitVLA	37.85	101.11	355.65	2199
Evo-1	63.60	131.01	458.84	2135
GR00T-N1.5	14.17	28.78	84.76†	5975
π_0	9.74	27.90	39.10†	6068
GR00T-N1.6	10.29	26.70	*	-
GR00T-N1.7	10.26	26.84	*	-

Over twenty trials per task across five workspace configurations, `vla.cpp` reaches $35/40$ (87.5%) against $16/40$ (40.0%) for the PyTorch baseline; the 95% Wilson intervals, $[73.9, 94.6]\%$ versus $[26.3, 55.4]\%$, do not overlap. The gap is widest on the longer-horizon Task 1, where the baseline times out under retry, and successful trials also finish faster (49 vs 71 s on Task 1; 28 vs 30 s on Task 2). Both effects follow the staleness mechanism: at ≈ 470 rather than ≈ 620 ms per chunk, each chunk is planned against an observation ≈ 150 ms fresher, and the advantage compounds across a long horizon.

V. CONCLUSION

We presented `vla.cpp`, a portable C++ runtime that serves VLA inference pattern for seven architectures from one binary and one bundle format, reproducing each reference checkpoint's behavior while running it faster than the released PyTorch implementation. The same bundle runs unchanged from a consumer GPU down to an 8 GB Orin Nano, where memory capacity, not latency, is the binding constraint, and a tensor-core ternary GEMM cuts BitVLA per-step latency by $4.5\times$. On a physical ALOHA arm, the runtime's lower per-chunk latency lets the policy replan against fresher observations, more than doubling closed-loop success over PyTorch.

VI. Limitations

While `vla.cpp` provides a highly portable, C++ runtime optimized for edge hardware, it has several key limitations. Notably, it lacks a dynamic weight-staging mechanism, which restricts its broad architectural support on highly resource-constrained devices like the 8 GB Jetson Orin Nano, where larger models exhaust the unified memory pool and require the simulator to be offloaded to an external machine. Furthermore, its native low-bit quantization is currently limited to the ternary configurations of BitVLA. Finally, the runtime reveals a high sensitivity to reduced-precision configurations (such as FP16) within multimodal encoders, where even minor vision-tower rounding discrepancies can silently compound across flow-matching solver steps and displace physical robot actions, necessitating strict build-time gating to avoid failure.

VII. AI Usage Statement

AI-based tools were used in a limited, assistive capacity during this work. Their use fell into three categories: (i) editing and polishing the writing of this manuscript, including rewording, grammar, and formatting, without introducing technical claims; (ii) refactoring existing source code of the runtime, with no architectural or algorithmic decisions delegated to the tools; and (iii) assisting with experiment tooling and analysis, such as scripting benchmark runs, parsing measurements, and generating figures. The runtime design, the numerical-parity methodology, the experimental protocol, and all reported results are the authors' own.

References

[1] Cheng Chi, Zhenjia Xu, Siyuan Feng, Eric Cousineau, Yilun Du, Benjamin Burchfiel, Russ Tedrake, and Shuran Song. Diffusion Policy: Visuomotor Policy Learning via Action Diffusion. In *Robotics: Science and Systems (RSS)*, 2023. arXiv:2303.04137.

[2] Senyu Fei, Siyin Wang, Junhao Shi, others, and Xipeng Qiu. LIBERO-Plus: In-depth Robustness Analysis of Vision-Language-Action Models. arXiv:2510.13626, 2025.

[3] Elias Frantar, Roberto L. Castro, Jiale Chen, Torsten Hoefler, and Dan Alistarh. MARLIN: Mixed-Precision Auto-Regressive Parallel Inference on Large Language Models. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2025. arXiv:2408.11743.

[4] Georgi Gerganov et al. llama.cpp. https://github.com/ggml-org/llama.cpp, 2023.

[5] Google. Protocol Buffers. https://protobuf.dev, 2008.

[6] iMatix. ZeroMQ: An Open-Source Universal Messaging Library. https://zeromq.org, 2007.

[7] Wenqi Jiang, Jason Clemons, Karu Sankaralingam, and Christos Kozyrakis. How Fast Can I Run My VLA? Demystifying VLA Inference Performance with VLA-Perf. *arXiv preprint arXiv:2602.18397*, 2026.

[8] Moo Jin Kim et al. OpenVLA: An Open-Source Vision-Language-Action Model. *arXiv preprint arXiv:2406.09246*, 2024.

[9] Woosuk Kwon et al. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proc. SOSP*, 2023. arXiv:2309.06180.

[10] Xuanlin Li et al. Evaluating Real-World Robot Manipulation Policies in Simulation. *arXiv preprint arXiv:2405.05941*, 2024. SimplerEnv.

[11] Bo Liu et al. LIBERO: Benchmarking Knowledge Transfer for Lifelong Robot Learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[12] Microsoft. BitBLAS: A Library to Support Mixed-Precision Matrix Multiplications for Quantized LLM Deployment. https://github.com/microsoft/BitBLAS, 2024.

[13] Microsoft. bitnet.cpp: Official Inference Framework for 1-bit LLMs. https://github.com/microsoft/BitNet, 2024.

[14] MINT-SJTU. Evo-1: Lightweight Vision-Language-Action Model with Preserved Semantic Alignment. *arXiv preprint arXiv:2511.04555*, 2025.

[15] MLC team. MLC-LLM: Universal LLM Deployment Engine with ML Compilation. https://llm.mlc.ai, 2024.

[16] NVIDIA. Isaac GR00T N1.x: Open Foundation Models for Generalist Robots. https://github.com/NVIDIA/Isaac-GR00T, 2025.

[17] NVIDIA. TensorRT Edge-LLM: An open-source c++ sdk for llm and vlm inference on edge platforms. https://github.com/NVIDIA/TensorRT-Edge-LLM, 2026. Accessed: 2026-05-28.

[18] NVIDIA. TensorRT-LLM: An open-source library for optimizing large language model inference. https://github.com/NVIDIA/TensorRT-LLM, 2026. Accessed: 2026-05-28.

[19] Physical Intelligence. π_0: A Vision-Language-Action Flow Model. https://github.com/Physical-Intelligence/openpi, 2024.

[20] Mustafa Shukor et al. SmolVLA: A Vision-Language-Action Model for Affordable and Efficient Robotics. *arXiv preprint arXiv:2506.01844*, 2025.

[21] Manoj Vishwanathan, Suvinay Subramanian, and Anand Raghunathan. Characterizing VLA Models: Identifying the Action Generation Bottleneck for Edge AI Architectures. arXiv:2603.02271, 2026.

[22] Hongyu Wang et al. BitVLA: 1-bit Vision-Language-Action Models for Robotics Manipulation. *arXiv preprint arXiv:2506.07530*, 2025.

[23] Jinheng Wang, Hansong Zhou, Ting Song, Shijie Cao, others, and Furu Wei. Bitnet.cpp: Efficient Edge Inference for Ternary LLMs. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2025. arXiv:2502.11880.

[24] Lei Wang, Lingxiao Ma, Shijie Cao, Quanlu Zhang, Jilong Xue, et al. Ladder: Enabling Efficient Low-Precision Deep Learning Computing through Hardware-aware Tensor Transformation. In *USENIX Symposium on*

Operating Systems Design and Implementation (OSDI), 2024.

[25] J. Williams, K. D. Gupta, R. George, and M. Sarkar. LiteVLA-Edge: Quantized On-Device Multimodal Control for Embedded Robotics. arXiv:2603.03380, 2026.

[26] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 52(4):65–76, 2009. doi: 10.1145/1498765.1498785.

[27] Xueyang Zhou, Yangming Xu, Guiyao Tie, et al. LIBERO-PRO: Towards Robust and Fair Evaluation of Vision-Language-Action Models Beyond Memorization. arXiv:2510.03827, 2025.

APPENDIX

A. From Architecture to Implementation

1) VLA inference pattern: `vla.cpp` runs as two processes (Figure 1). A C++ server loads the model once and holds it resident; on each request it decodes an observation (camera images, language instruction, proprioceptive state) from a Protobuf message, executes the inference pattern, denormalizes the actions into robot units using statistics carried in the bundle, and returns the chunk. The server is deliberately *stateless*: control-history concerns such as blending overlapping chunks, handling stale observations, and safe-holding when inference is late reside on the client, typically Python for interoperability with existing robotics and simulator code. The two sides communicate over ZeroMQ [6] with a Protobuf [5] schema, chosen for low loopback latency, language-agnostic bindings, and a typed contract generated from one schema file. Each model is a single GGUF bundle holding the language model, action head, normalization statistics, and architecture configuration; the vision encoder is embedded or supplied as a companion multimodal file.

2) The served architectures: Table III 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 III
THE SEVEN SERVED ARCHITECTURES. FM = FLOW-MATCHING; ALTVL = ALTERNATEVL. 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	Language	Action head	Par.	T
SmolVLA	SigLIP-So400m	SmolLM2-360M	FM cross-attn	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	3B	10
GR00T-N1.5	SigLIP2-400M	Qwen3-1.7B	AltVL DiT+s.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+s.attn	3B	4

Table IV reports success, latency, and footprint for all seven architectures on the consumer GPU. Every architecture reproduces the behavior of its reference checkpoint. BitVLA reaches a perfect 200 of 200 episodes. This model is also the closest-matching architecture that differs from its reference Python checkpoint by a single episode out of 200. These results confirm that the engine is faithful to the reference implementations rather than merely fast. The per-task, per-episode comparison against each reference checkpoint, the reference-checkpoint provenance (exact weights, configuration, and harness), and the SimplerEnv (WidowX) results appear in appendices D and G.

Success tracks chunk length and backbone rather than raw forward cost: π_0 is the cheapest per step yet the least accurate, since replaying many actions between observations lets the scene drift from the plan. Our claim is behavioral *parity* with the reference, not new task capability, so a near-saturated suite is still informative: its low variance makes a one-episode gap meaningful, even though such scores are known to overstate generalization [27, 2]. Appendix D gives the full Spatial, Goal, and Long suites with Wilson intervals and the per-task, per-episode comparison against each reference.

B. Cross-hardware roofline and the utilization lever

BitVLA makes the separation between footprint and speed especially clear. Packing its ternary weights into a two-bit layout cuts the on-disk model from 5.6 to 1.34 GiB and the load-time high-water mark from 11.1 to 1.15 GiB, enough to load on the Nano, yet leaves latency unchanged: both layouts feed identical values to the same kernel. The roofline [26] (Figure 2) explains why. A single observation is one forward pass over several hundred prefix tokens, not a token-by-token decode: each backbone weight is reused across those tokens, so the prefix computation is firmly compute-bound, far from the bandwidth wall. The low-token action expert that consumes the prefix is, by contrast, memory-bound; consistent with VLA-Perf [7] and independent edge characterizations [21], its arithmetic intensity is low and its wall-clock share grows with the solver-step count. In our deployed configurations, with a several-hundred-token prefix over few steps, the compute-bound prefix dominates the forward, the regime this roofline characterizes. We plot the GDDR-backed RTX 3060 and AGX Orin; on the LPDDR Orin Nano it is the 8 GB *capacity*, not bandwidth, that bounds which models load, so we confine the prefix-compute-bound statement to the GDDR tiers. Ternary quantization shrinks bytes per weight, not multiply–accumulate operations: it lets the model fit, with no speedup of its own.

The choice of arithmetic unit, not weight format, then determines latency. The reference `bitnet.cpp` GPU kernel [13, 23] runs its W2A8 (two-bit weight, eight-bit activation) dot products on the CUDA cores via the `dp4a` intrinsic, which leaves idle the dedicated integer matrix-multiply–accumulate (IMMA) tensor cores, the units that deliver peak throughput on our devices. We rebuilt BitVLA's ternary matmul to run on the

TABLE IV

BEHAVIORAL FIDELITY AND SINGLE-REQUEST PROFILE ON THE RTX 3060, FULL LIBERO_OBJECT SUITE (200 EPISODES/ARCH). *step*: WALL-CLOCK PER ENVIRONMENT STEP (AMORTIZED OVER CHUNK REPLAY); *inf*: SERVER-SIDE FORWARD; n_a: CHUNK LENGTH; BRACKETS ARE 95% WILSON INTERVALS.

Model	Backbone	n_a	SR (%)	step (ms)	inf (ms)	VRAM (MiB)
SmolVLA	SmolVLM2	4	90.5 [86, 94]	28.16	54.8	1410
π_0	PaliGemma	32	87.5 [82, 91]	9.74	207.2	5548
BitVLA	BitNet–SigLIP	8	**100.0** [98, 100]	37.85	235.9	**1312**
Evo-1	InternVL3	8	94.5 [90, 97]	63.60	131.0	1564
GR00T-N1.5	Eagle/Cosmos	16	96.0 [92, 98]	14.17	147.0	4866
GR00T-N1.6	Eagle/Cosmos	16	86.5 [81, 91]	10.29	83.6	6048
GR00T-N1.7	Eagle/Cosmos	16	98.0 [95, 99]	10.26	84.1	6302

IMMA tensor cores while reading the same packed weights. Tensor-core kernels for low-bit weights are well studied on server GPUs [24, 12, 3]; our contribution is a *portable*, `ggml`-class W2A8 tensor-core GEMM that serves a VLA action expert unchanged across all three tiers, against the `dp4a` path that is the de facto ternary baseline. It cuts BitVLA per-step latency from 172.8 to 37.85 ms on the RTX 3060 ($4.6\times$) and 406.6 to 101.11 ms on the AGX Orin ($4.0\times$), and $4.5\times$ in isolation, with bit-identical outputs and unchanged 100% success. Moving the same operations onto higher-throughput units recovers a $4\times$ gain that no weight-layout change could deliver, the prediction that follows when utilization, not bandwidth, is the bottleneck for this kernel. Appendix F gives the per-component breakdown (vision, language, action expert) and anchors each operating point to the analytical roofline.

C. Implementation of the Prefix Extension

The runtime targets the two-stage structure: 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 $[\text{tokens} \times d]$ 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 B, exercising the footprint and kernel axes rather than the iterative-head axis of the other six.

D. Per-Task Agreement Across LIBERO Suites

Table V 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 A2, 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 V

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]

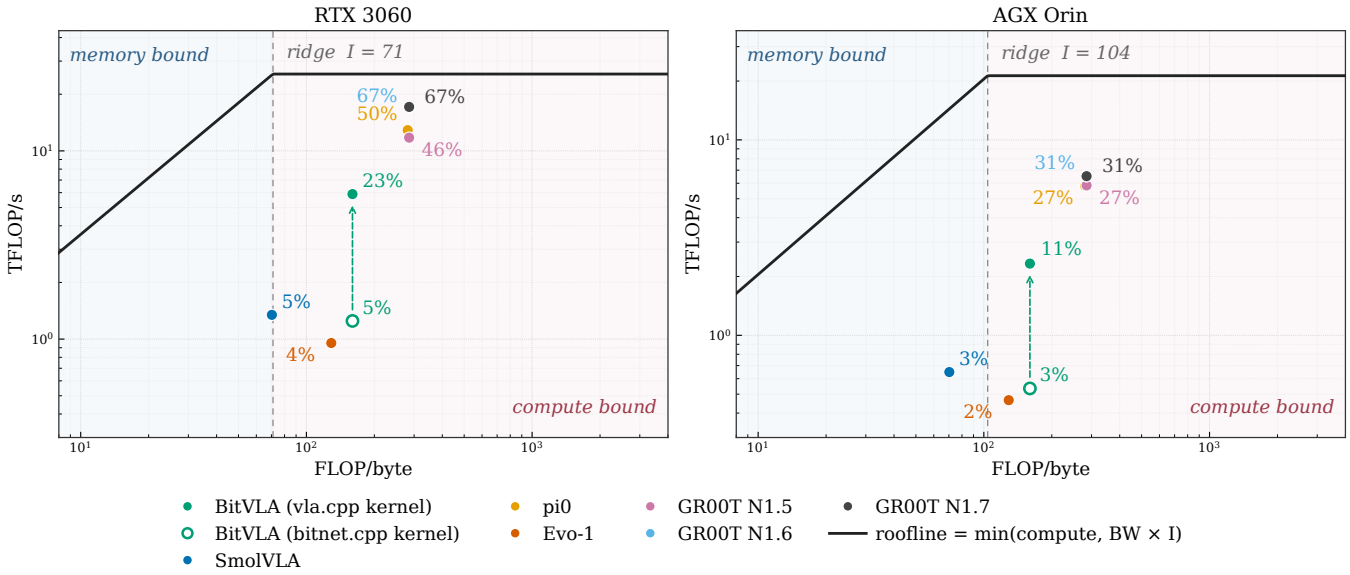

Fig. 2. Roofline placement of single-request VLA inference (LIBERO-Object sweep) on the RTX 3060 (sm_86; 25.6 TFLOP/s FP16, 360 GB/s) and Jetson AGX Orin (sm_87; 21.3 TFLOP/s, 204.8 GB/s). Each dot is one model at its operational intensity (FLOPs/byte) versus achieved throughput (estimated FLOPs over measured server-side time per call); the label is the achieved fraction of device peak, and the gap to the roof is the remaining utilization headroom.

E. Optimized and Cross-Stack Baselines

This section discusses the speedup of `vla.cpp` compared to the eager-mode PyTorch reference and the graph-captured PyTorch configuration across three models and two devices (Table VI). 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 the Table I records why each stack does not provide an apples-to-apples baseline for this inference pattern.

TABLE VI
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. A DASH MARKS AN ARCHITECTURE FOR WHICH NO FULLY GRAPH-CAPTURED CONFIGURATION IS AVAILABLE.

Model	Device	PyTorch (eager)	PyTorch (graph-cap.)	`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**

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). 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 VII.

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

TABLE VII
CLOSED-LOOP PERFORMANCE ON THE ALOHA LEFT ARM WITH GR00T-N1.6 IN BF16. BOTH ENGINES RUN THE SAME CHECKPOINT WITH AN EIGHT-STEP PREDICTION CHUNK OF WHICH SEVEN ACTIONS ARE EXECUTED BEFORE REPLANNING.

	Task 1: blocks → box		Task 2: trash + banana		
Engine	SR	Task time	SR	Task time	Inference / chunk
PyTorch baseline	3/20 (15%)	71 s	13/20 (65%)	30 s	∼620 ms
`vla.cpp`	**18/20 (90%)**	**49** s	**17/20 (85%)**	**28** s	∼**470** ms

this architecture remains its deployment footprint, portability across the Orin tiers, and the ternary and flow-expert paths it uniquely supports.

F. Per-Component Roofline Breakdown

Table VIII 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 B.

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 IX. 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 (Table IX). 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.

TABLE VIII
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 ($\approx$ 5.4 MS/STEP MEASURED).

Phase	Lat. (%)	Op. int. (F/B)	Regime
Prefix (vision + LM)	75	∼256-530	comp.-bound
Action expert	25	∼50	mem.-bound
Total forward path	100	-	prefix-dom.

Table IX makes the model-dependence concrete. 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 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. B); each phase's operational intensity is not.

TABLE IX
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 III); THE SINGLE-PASS BITVLA HEAD IS MARKED −. THE PREFIX PHASE COMBINES THE VISION ENCODER AND THE LANGUAGE-MODEL PREFILL.

Model	Lang. backbone	Size	T	Prefix	Act. exp.
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.

G. 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.

H. Replan Interval and the Latency-Staleness Trade-off

The closed-loop argument of Section IV-B 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

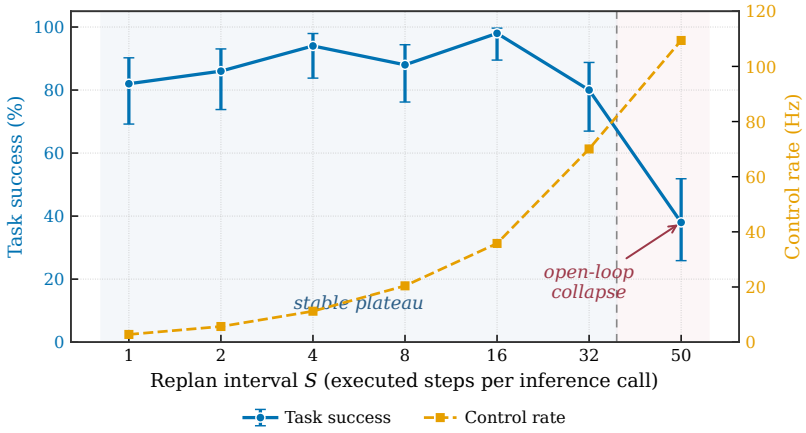

Fig. 3. Replan-interval study for SmolVLA on `libero_object` (Jetson Orin Nano, 50 episodes/setting). Task success (left axis, 95% Wilson intervals) is stable across the shaded plateau and collapses once execution becomes fully open-loop at S=50. The amortized control rate (right axis) rises almost linearly with S.

TABLE X
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. med. (ms)	Lat. p95 (ms)	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]

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.

Table X 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 3 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 IV-B 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 IV: a long action chunk consumed without re-planning drifts from the observation it was conditioned on. The practical guidance is to replan within the plateau rather than at the chunk boundary.