# QFAST: FAST QUANTIZED INFERENCE FOR ROBOTICS VISION-LANGUAGE-ACTION MODELS

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Vision-Language-Action (VLA) models have demonstrated strong performance in robotic manipulation by unifying visual perception, language understanding, and action prediction within transformer-based architectures. However, their deployment on resource-constrained edge devices faces critical challenges: inference latency and memory requirements that prevent real-time operation. Existing quantization methods prioritize accuracy preservation and memory compression while largely neglecting inference speed—a bottleneck for real-time robotic control requiring sub-100ms latency. We present **QFast**, a quantization inference system designed to accelerate VLA models on edge devices. QFast employs speed-oriented quantization that balances accuracy and latency through module-specific bit-width selection, per-token weight quantization with dynamic per-channel activation quantization, and selective preservation of precision-sensitive layers. To exploit hardware capabilities, we implement optimized GEMM kernels tailored to diverse VLA matrix dimensions, enabling native low-bit computation on Tensor Cores. Evaluated on Pi0.5 and OpenVLA, QFast achieves 1.3× end-to-end speedup with up to 3.6× memory reduction while maintaining task performance on LIBERO benchmarks. QFast represents the first quantization system enabling hardware-accelerated low-bit inference for VLA models, making real-time edge deployment practical. Code is available at `https://anonymous.4open.science/r/QFast-D157/`.

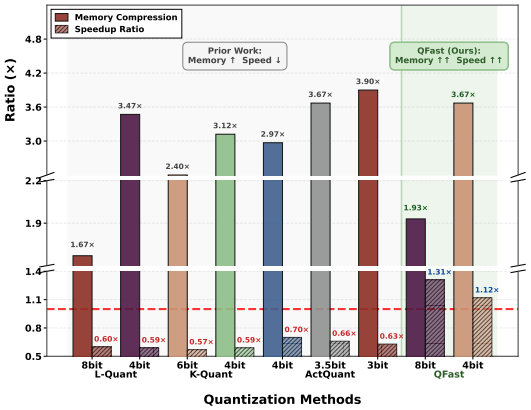
## 1 INTRODUCTION

Vision-Language-Action (VLA) models have demonstrated remarkable capabilities in robotic manipulation by unifying visual perception, language understanding, and action prediction within a single transformer-based architecture Kim et al. (2024); Brohan et al. (2023); Black et al. (2024); Team et al. (2024). These models leverage pre-trained vision-language representations to generalize across diverse tasks and environments with minimal task-specific fine-tuning. However, their deployment on resource-constrained robotic platforms faces a critical bottleneck. Real-time manipulation requires inference latency below 100ms to maintain stable 10Hz control loops Brohan et al. (2022). Current VLA models contain billions of parameters and occupy multi-gigabyte memory footprints. This scale prevents deployment on edge devices that must satisfy these real-time constraints. This gap demands efficient techniques that preserve task performance while satisfying strict constraints on model size, memory bandwidth, and power consumption.

Model quantization has emerged as the primary approach for deploying large models on edge devices. However, existing quantization methods for VLA models exhibit a critical failure. As Figure 1 (left) reveals, they achieve 3.9× memory compression yet *degrade inference speed to* 0.63 *of the FP16 baseline*. This counterintuitive result stems from a fundamental misalignment in optimization objectives. Prior work Akbari et al. (2026); Xu et al. (2026); Wang et al. (2025) prioritizes aggressive memory compression. They employ fine-grained quantization strategies and mixed-precision schemes to preserve accuracy. Yet these techniques introduce substantial dequantization overhead during inference. More critically, without genuine low-bit arithmetic on hardware accelerators, quantized models merely reduce memory footprint at the cost of substantial dequantization overhead. They still perform computation in higher precision after dequantization, negating any speed benefit and often running slower than FP16 baselines.



Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.




| Method | Training-Free | Low-bit Tensor Core | Acceler-ation | Quantized Components | | |
|---|---|---|---|---|---|---|
| | | | | Vision | Language | Action |
| QAIL | ✗ | ✗ | ✗ | ✓ | ✓ | ✗ |
| BitVLA | ✗ | ✗ | ✗ | ✓ | ✓ | ✗ |
| L-Quant | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ |
| K-Quant | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ |
| QVLA | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ |
| QuantVLA | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ |
| ActQuant | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ |
| **QFAst (Ours)** | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

Figure 1: **Prior VLA quantization methods achieve memory compression but fail to accelerate inference. Left**: Memory-speed trade-off. Prior methods (L-Quant, K-Quant, ActQuant) achieve $2.58$–$2.81\times$ compression yet suffer $0.85$–$0.92\times$ speedup. QFast delivers $1.93$–$3.40\times$ compression with $1.22$–$1.31\times$ actual speedup via hardware-aligned quantization. **Right**: Quantization coverage. Prior methods selectively quantize language/vision modules without Tensor Core utilization, forcing mixed-precision execution. QFast comprehensively quantizes all components with native INT4/INT8 kernels, enabling end-to-end acceleration.

Three root causes underlie this speed degradation. First, existing methods suffer from misaligned hardware utilization. Methods like ActQuant Akbari et al. (2026) use specialized inference frameworks but fail to exploit low-bit Tensor Core instructions (INT4/INT8), leaving the most powerful computational units idle. Second, incomplete quantization coverage prevents end-to-end acceleration. Figure 1 (right) shows that prior methods selectively quantize only language and vision modules while keeping action heads in FP16, forcing mixed-precision execution that bottlenecks the entire pipeline. Third, accuracy-preservation strategies introduce substantial overhead. Extremely low bit-widths (3–4 bits) and fine-grained granularity require complex dequantization logic that negates any speedup from reduced data movement. The result is a paradox: models that occupy less memory but run slower than their uncompressed counterparts.

We present **QFast**, a speed-first quantization system that shifts the optimization objective from memory compression to inference acceleration through hardware-software co-design. The key insight is that moderate bit-widths aligned with Tensor Core capabilities (INT4/INT8) enable genuine low-bit computation, while comprehensive module coverage eliminates mixed-precision bottlenecks. QFast addresses three technical challenges: (1) profiling heterogeneous VLA components to determine optimal per-module bit-widths that maximize overall throughput; (2) implementing adaptive kernel selection to handle variable matrix dimensions from large language backbones to small action heads; (3) developing optimized INT4/INT8 GEMM kernels with fused dequantization and IMMA instruction scheduling that enable authentic low-bit computation. Evaluated on $\pi_0$ Black et al. (2024) and OpenVLA Kim et al. (2024), QFast achieves up to $1.80\times$ kernel-level speedup and $1.31\times$ end-to-end acceleration with $<2\%$ accuracy loss, delivering $1.93$–$3.40\times$ memory compression. To our knowledge, QFast is the first quantization system that achieves true hardware-accelerated inference for VLA models through genuine low-bit Tensor Core computation.

**Contributions.** This work makes the following contributions:

- Identification and analysis of the speed degradation problem in existing VLA quantization methods, revealing the fundamental tension between memory optimization and inference acceleration.
- A speed-first quantization system that shifts the design paradigm from accuracy preservation to throughput maximization through hardware-software co-design.
- A systematic module-specific profiling methodology and adaptive kernel selection mechanism tailored to the heterogeneous computational patterns of VLA architectures.
- Optimized INT4/INT8 GEMM kernels with fused dequantization and Tensor Core utilization, achieving $1.22$–$1.31\times$ end-to-end speedup—the first demonstration of actual acceleration for quantized VLA models.

## 2 Related Work

**Vision-Language-Action Models.** VLA models unify visual perception, language understanding, and robotic control within a single framework Kim et al. (2024); Brohan et al. (2023); Black et al. (2024); Team et al. (2024). Representative works include OpenVLA Kim et al. (2024), which adapts Prismatic VLM for manipulation, RT-2 Brohan et al. (2023) that leverages web-scale vision-language pre-training, and Pi0 Black et al. (2024) with flow-matching action prediction. While demonstrating strong generalization across manipulation tasks, their billion-parameter scale creates deployment challenges on edge devices.

**Model Quantization.** Post-training quantization methods for large models focus primarily on memory reduction Frantar et al. (2022); Lin et al. (2024); Xiao et al. (2023); Shao et al. (2024). GPTQ Frantar et al. (2022) applies layer-wise quantization based on approximate second-order information, while AWQ Lin et al. (2024) protects salient weights identified by activation magnitudes. SmoothQuant Xiao et al. (2023) migrates quantization difficulty from activations to weights through channel-wise scaling, and OmniQuant Shao et al. (2024) optimizes both quantization parameters and weight clipping simultaneously. For vision-language models, VLMQ Xue et al. (2025) and VLM-PTQ Deng & Huang (2026) address modality-specific challenges through token saliency and asymmetric quantization. However, these methods prioritize accuracy preservation over inference speed and often rely on dequantization during computation.

**VLA Quantization.** Recent work addresses VLA-specific quantization challenges. QuantVLA Zhang et al. (2026) introduces scale-calibrated PTQ for diffusion transformer action heads but focuses on memory compression rather than speed. QVLA Xu et al. (2026) proposes channel-wise mixed-precision allocation based on action-space sensitivity, achieving aggressive compression but with irregular bit-widths that complicate hardware acceleration. ActQuant Akbari et al. (2026) introduces action-guided quantization reaching sub-4-bit compression and develops OmniModel.cpp for deployment. However, none of the aforementioned methods fully leverage the native INT4/INT8 Tensor Core computing capabilities, thereby limiting actual acceleration performance. Due to dequantization, these methods often run slower than FP16 counterparts.

**Efficient Inference Systems.** High-performance inference systems have been developed for large models Kwon et al. (2023); Dao et al. (2022); NVIDIA Corporation (2023). vLLM Kwon et al. (2023) introduces PagedAttention for efficient memory management, while FlashAttention Dao et al. (2022) optimizes attention computation through IO-aware tiling. llama.cpp Gerganov (2023) enables efficient inference on CPUs and edge devices through optimized quantization and minimal dependencies. TensorRT-LLM NVIDIA Corporation (2023) provides optimized kernels including INT8 support, and libraries like CUTLASS NVIDIA Corporation (2017) enable efficient low-precision GEMM implementations. However, VLA currently lacks a unified inference framework, with each model requiring its own specific environment and dependencies.

**Our Work.** QFast is the first speed-oriented quantization system for VLA models that achieves true hardware-accelerated inference through INT4/INT8 Tensor Core operations. We shift the optimization objective from compression to speed while maintaining competitive accuracy.

## 3 Methodology

### 3.1 Overview

We present an end-to-end low-bit quantization framework for vision-language-action (VLA) policies that supports diverse architectural configurations. As illustrated in Figure 2, the framework accommodates models with separate vision encoders, language backbones, action experts, and action decoders, enabling quantization of the complete inference pipeline from multimodal input to action prediction.

The framework employs a mixed-precision strategy that balances computational efficiency with numerical stability. Linear projections are quantized to configurable low-bit representations (W8A8, W4A8, or W4A4), while normalization layers, biases, embeddings, and residual connections remain in floating-point precision (FP16 or FP32). After each quantized matrix multiplication, outputs are

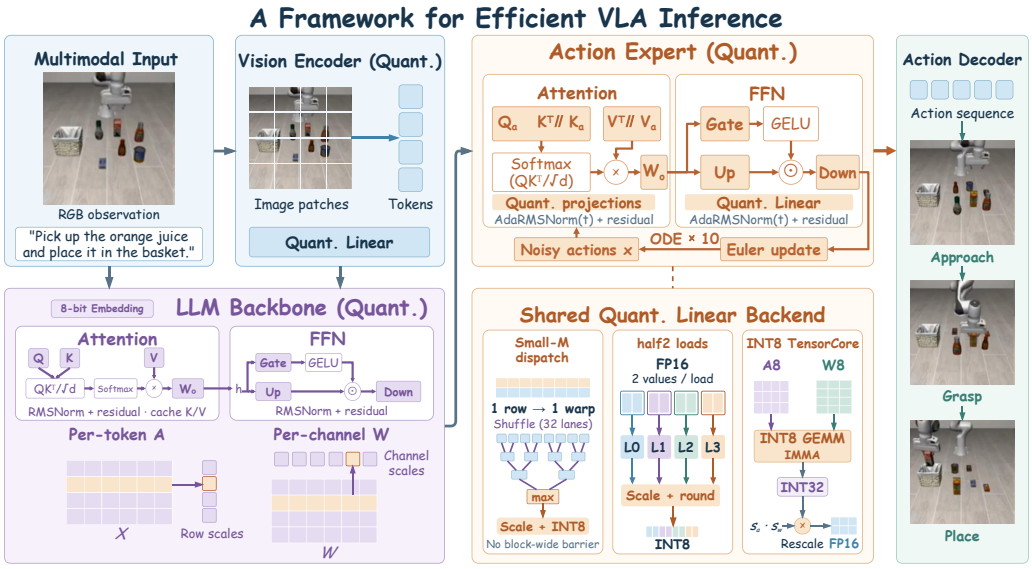



Figure 2: End-to-end low-bit inference framework for vision-language-action policies. Linear projections use configurable low-bit weights and activations (blue), while precision-sensitive operations remain in FP16/FP32 (orange).

dequantized and rescaled before feeding subsequent operators. Throughout this section, W$b_w$A$b_a$ denotes $b_w$-bit weights and $b_a$-bit activations unless otherwise specified.

### 3.2 Weight Quantization

We apply symmetric per-channel quantization to weight matrices. For a weight matrix $W \in \mathbb{R}^{N\times K}$ with $N$ output channels and $K$ input channels, the quantization range at bit width $b_w$ is defined by

$$q_w^{\max} = 2^{b_w-1} - 1. \tag{1}$$

For each output channel $n$, we compute a channel-specific scale based on the maximum absolute weight value:

$$s_n^w = \frac{\max\limits_{0\le k<K} |W_{n,k}|}{q_w^{\max}}. \tag{2}$$

Quantization then proceeds via rounding and clipping:

$$\widehat{W}_{n,k} = \mathrm{clip}\left(\mathrm{round}\left(\frac{W_{n,k}}{s_n^w}\right), -q_w^{\max}, q_w^{\max}\right), \tag{3}$$

with dequantization given by

$$W_{n,k} \approx s_n^w \widehat{W}_{n,k}. \tag{4}$$

Weight quantization is performed offline during model export. The quantized integer payload $\widehat{W}$ and corresponding FP32 scale vector $\mathbf{s}^w \in \mathbb{R}^N$ are stored together in the exported model.

### 3.3 Dynamic Activation Quantization

Activation quantization is performed at runtime to accommodate the varying distributions of visual tokens, language tokens, and action tokens. Given an FP16 activation matrix $X \in \mathbb{R}^{M\times K}$, where

$M$ represents the number of tokens or sequence elements, we compute a separate scale for each row. The quantization range at bit width $b_a$ is

$$q_a^{\max} = 2^{b_a - 1} - 1. \tag{5}$$

For row $m$, the activation scale is computed as

$$s_m^a = \frac{\max\limits_{0 \le k < K} |X_{m,k}|}{q_a^{\max}}. \tag{6}$$

Quantization follows the same symmetric scheme as weights:

$$\widehat{X}_{m,k} = \text{clip}\left(\text{round}\left(\frac{X_{m,k}}{s_m^a}\right), -q_a^{\max}, q_a^{\max}\right). \tag{7}$$

This per-token (per-row) quantization strategy adapts dynamically to the distinct statistical properties of different modalities within the VLA architecture. Activation quantization is implemented using optimized GPU kernels described in Appendix A.

### 3.4 Low-Bit Matrix Multiplication and Dequantization

The quantized linear transformation is computed as

$$C = \widehat{X}\widehat{W}^{\mathsf{T}}, \tag{8}$$

where matrix multiplication operates on low-bit integer operands with an accumulation precision determined by the hardware backend (typically INT32 for INT8 operands). The floating-point output is reconstructed by applying both activation and weight scales:

$$Y_{m,n} = s_m^a s_n^w C_{m,n} + b_n, \tag{9}$$

where $b_n$ is an optional FP32 bias term. The result is converted to FP16 or FP32 as required by downstream operations.

Kernel selection is hardware-dependent: INT8 configurations use Tensor Core kernels via cuBLASLt, while 4-bit configurations employ specialized packed-integer kernels with custom dequantization routines.

### 3.5 Implementation Optimizations

To achieve efficient inference across VLA architectures with heterogeneous computational patterns, we implement three kernel-level optimizations targeting distinct bottlenecks: small-batch action prediction, large-scale multimodal processing, and repeated transformer layer execution.

**Warp-Level Quantization for Small Batches.** Standard activation quantization assigns one CUDA thread block (128–256 threads) per input row, computing the row-wise maximum via shared memory reduction and block-wide synchronization. For VLA action experts processing small batches, this incurs severe under-utilization: only 4 active blocks on streaming multiprocessors (SMs) capable of scheduling 16+ blocks concurrently.

We deploy a specialized warp-level kernel where one warp (32 threads) quantizes each activation row. Thread $t$ processes a strided channel subset

$$\mathcal{I}_t = \{k : k \equiv t \pmod{32},\ 0 \le k < K\} \tag{10}$$

and computes the local maximum

$$r_{m,t} = \max_{k \in \mathcal{I}_t} |X_{m,k}|. \tag{11}$$

The row-wise scale is obtained via warp shuffle reduction:

$$a_m = \max_{t=0}^{31} r_{m,t}, \quad s_m^a = \frac{a_m}{q_a^{\max}}, \tag{12}$$

followed by in-warp quantization without global synchronization. This eliminates shared memory overhead and block barriers, significantly improving SM occupancy for action prediction pathways. The kernel activates automatically when $M \le 4$.

**Vectorized Memory Operations.** After INT8 GEMM produces the INT32 accumulator $C \in \mathbb{Z}^{M\times N}$, dequantization applies per-row activation scales and per-channel weight scales. A naive scalar kernel processes one element per thread:

$$Y_{m,n} = s^a_m \cdot s^w_n \cdot C_{m,n} + b_n, \tag{13}$$

generating many small memory transactions. Our optimized kernel processes four consecutive output channels per thread using vector instructions. For output row $m$ and channel group starting at $n$:

$$\mathbf{c}_{m,n:n+3} \xleftarrow{\texttt{int4}} C[m, n:n+3], \quad \mathbf{s}^w_{n:n+3} \xleftarrow{\texttt{float4}} \mathbf{s}^w[n:n+3], \tag{14}$$

$$\mathbf{y}_{m,n:n+3} = \mathbf{c}_{m,n:n+3} \odot (s^a_m \cdot \mathbf{s}^w_{n:n+3}) + \mathbf{b}_{n:n+3}, \tag{15}$$

where $\odot$ denotes element-wise multiplication. Results are packed as `half2` (FP16) or `float4` (FP32). Non-divisible dimensions use scalar boundary kernels for remainder channels.

For activation quantization in vision encoders and language backbones, we employ vectorized `half2` loads when $K$ is even:

$$(X_{m,k}, X_{m,k+1}) \xleftarrow{\texttt{half2}} X[m, k:k+1], \tag{16}$$

halving memory transactions and improving bandwidth utilization for large activation matrices ($M \geq 256$).

**Descriptor Caching and Memory Layout.** Creating cuBLASLt descriptors and heuristic plans incurs non-trivial host overhead. We maintain a cache indexed by matrix shape $(M, N, K)$, amortizing this cost across transformer layers with repeated dimensions. Weight matrices are stored in row-major $[N, K]$ format where each row corresponds to one output channel:

$$\widehat{W} = \begin{bmatrix} \widehat{W}_{0,0} & \cdots & \widehat{W}_{0,K-1} \\ \vdots & \ddots & \vdots \\ \widehat{W}_{N-1,0} & \cdots & \widehat{W}_{N-1,K-1} \end{bmatrix}, \quad \mathbf{s}^w = \begin{bmatrix} s^w_0 \\ \vdots \\ s^w_{N-1} \end{bmatrix}. \tag{17}$$

This layout ensures contiguous memory access during per-channel scale application, maximizing cache locality during dequantization.

These optimizations collectively reduce kernel launch overhead, synchronization costs, and memory access latency without altering the quantization semantics defined in Equations 3–9. Comprehensive CUDA kernel specifications, cuBLASLt configuration parameters, and performance characterization across matrix dimensions are provided in Appendix A.

# 4 Experiments

## 4.1 Experimental Setup

**Models.** We evaluate QFast on two state-of-the-art vision-language-action (VLA) models: $\pi_{0.5}$ (Black et al., 2024) (0.5B parameters) and OpenVLA-OFT (Kim et al., 2024) (7B parameters), both pretrained on large-scale robot manipulation datasets and fine-tuned on LIBERO tasks.

**Benchmark.** We conduct experiments on the LIBERO benchmark (Liu et al., 2023), which contains four diverse manipulation suites: (1) *Spatial*: spatial relationship reasoning; (2) *Object*: object-centric manipulation; (3) *Goal*: goal-conditioned tasks; (4) *LIBERO-10*: long-horizon compositional tasks. Each suite contains 100 evaluation episodes, totaling 400 episodes per model configuration.

**Baselines.** We compare QFast against three quantization baselines: L-Quant (blockwise quantization, $Q8_0$/$Q4_0$), K-Quant (super-block quantization, $Q8_K$/$Q4_K$/$Q6_K$), and ActQuant (Akbari et al., 2026) (mixed-precision activation-aware quantization, 2.5-4.0 bits). All baselines use official implementations from `llama.cpp` (Gerganov, 2023) and the original ActQuant repository.

**Metrics and Implementation.** We report success rate (SR, %), memory usage (GB), and wall-clock speedup relative to FP16 baseline inference. QFast employs dynamic group sizes $g \in \{32, 64, 128\}$ calibrated on 512 demonstration trajectories from the LIBERO training set. All experiments were conducted on NVIDIA A800-SXM4-80GB GPUs (8.0 Ampere architecture).

Table 1: End-to-end performance of $\pi_{0.5}$ and OpenVLA-OFT on the LIBERO benchmark.

| Model | Method | Bit | Mem. | Spatial | | Object | | Goal | | LIBERO-10 | | Avg. | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | SR | Speedup | SR | Speedup | SR | Speedup | SR | Speedup | SR | Speedup |
| $\pi_{0.5}$ | FP16 | 16 | 6.2 | 99% | 1.00× | 97% | 1.00× | 92% | 1.00× | 97% | 1.00× | 96.2% | 1.00× |
| | L-Quant. | 8 | 3.7 | 99% | 0.83× | 97% | 0.85× | 90% | 0.86× | 96% | 0.85× | 95.5% | 0.85× |
| | | 4 | 2.3 | 97% | 0.88× | 97% | 0.87× | 83% | 0.86× | 94% | 0.86× | 92.7% | 0.87× |
| | K-Quant. | 8 | 3.9 | 99% | 0.92× | 97% | 0.90× | 91% | 0.89× | **96%** | 0.90× | **95.7%** | 0.90× |
| | | 4 | 2.3 | 98% | 0.84× | 97% | 0.84× | 88% | 0.83× | 93% | 0.84× | 94.0% | 0.85× |
| | ActQuant | 4.0 | 2.4 | **99%** | 0.92× | 96% | 0.93× | 90% | 0.92× | 93% | 0.93× | 94.5% | 0.92× |
| | | 3.5 | 2.3 | 99% | 0.89× | 99% | 0.89× | **94%** | 0.88× | 92% | 0.88× | **96.3%** | 0.88× |
| | | 3.0 | 2.2 | 99% | 0.81× | 96% | 0.86× | 90% | 0.87× | 93% | 0.85× | 94.5% | 0.85× |
| | **QFast** | 8 | **3.2(1.8x)** | **99%** | **1.22**× | **97%** | **1.23**× | 89% | **1.21**× | **96%** | **1.20**× | 95.2% | **1.22**× |
| | | 4 | **1.8(3.4x)** | 98% | **1.16**× | **97%** | **1.18**× | 86% | **1.19**× | **94%** | **1.18**× | 93.7% | **1.18**× |
| OpenVLA-OFT | FP16 | 16 | 12.5 | 98% | 1.00× | 97% | 1.00× | 96% | 1.00× | 97% | 1.00× | 97.0% | 1.00× |
| | L-Quant. | 8 | 6.7 | 99% | 0.61× | 99% | 0.60× | **97%** | 0.61× | 96% | 0.60× | 97.7% | 0.60× |
| | | 4 | 3.6 | 98% | 0.60× | 98% | 0.59× | 94% | 0.59× | 96% | 0.58× | 96.5% | 0.59× |
| | K-Quant. | 6 | 5.2 | 98% | 0.56× | 96% | 0.57× | 96% | 0.58× | 98% | 0.56× | 97.0% | 0.57× |
| | | 4 | 3.9 | **98%** | 0.60× | 97% | 0.59× | 96% | 0.58× | 97% | 0.59× | **96.7%** | 0.59× |
| | ActQuant | 4.0 | 4.2 | 96% | 0.70× | 98% | 0.71× | 97% | 0.70× | 95% | 0.71× | 96.6% | 0.70× |
| | | 3.5 | 3.4 | 95% | 0.68× | 98% | 0.65× | 96% | 0.66× | 95% | 0.67× | 96.5% | 0.66× |
| | | 3.0 | 3.2 | 92% | 0.62× | 97% | 0.63× | 94% | 0.64× | 95% | 0.64× | 95.0% | 0.63× |
| | | 2.5 | 2.7 | 86% | 0.60× | 98% | 0.59× | 84% | 0.61× | 91% | 0.61× | 90.1% | 0.60× |
| | **QFast** | 8 | **6.4(1.9**×**)** | **99%** | **1.32**× | **99%** | **1.29**× | 94% | **1.33**× | **99%** | **1.31**× | **97.7%** | **1.31**× |
| | | 4 | **3.4(3.6**×**)** | 97% | **1.12**× | **98%** | **1.14**× | 94% | **1.11**× | 93% | **1.12**× | 95.5% | **1.12**× |

*Notes.* Each LIBERO suite contains 100 evaluation episodes per configuration. Average success rates are computed over 400 episodes. SR denotes success rate (%). Speedup denotes mean server-side policy inference speed relative to *FP16*, rather than end-to-end environment throughput. Bold values indicate the best result in each column among the available configurations. L-Quant. denotes the original blockwise formats ($Q4_0$ and $Q8_0$), whereas K-Quant. denotes the K-quant super-block formats ($Q4_K$ and $Q8_K$).

### 4.2 End-to-End VLA Model Performance

Table 1 evaluates QFast against existing quantization methods on the LIBERO benchmark suite (Liu et al., 2023). We test two VLA models ($\pi_{0.5}$ and OpenVLA-OFT) across four manipulation suites (*Spatial*, *Object*, *Goal*, *LIBERO-10*), totaling 400 episodes per configuration.

**QFast achieves genuine acceleration under compression.** At 8-bit, QFast delivers $1.22\times$ ($\pi_{0.5}$) and $1.31\times$ (OpenVLA-OFT) speedups with $1.8$–$1.9\times$ memory reduction, maintaining success rates within 1 percentage point of FP16 (95.2% vs. 96.2%; 97.7% vs. 97.0%). At 4-bit, speedups remain at $1.12$–$1.18\times$ under $3.4$–$3.6\times$ compression, with accuracy degrading only 1.5–2.5 percentage points. Prior methods introduce severe slowdowns: L-Quant ($0.60$–$0.85\times$), K-Quant ($0.57$–$0.90\times$), and ActQuant ($0.63$–$0.92\times$), rendering them impractical for real-time control despite comparable compression ratios. QFast is the only method achieving memory reduction, inference acceleration, and accuracy preservation simultaneously.

**Speedups remain consistent across tasks and bit-widths.** Across LIBERO suites, QFast maintains $1.20$–$1.23\times$ speedups on $\pi_{0.5}$ and $1.29$–$1.33\times$ on OpenVLA-OFT at 8-bit, with task success rates matching or approaching FP16 (97–99% on most suites). At 4-bit, speedups persist at $1.11$–$1.19\times$ with accuracy ranging 86–98%, demonstrating robustness across bit-widths and task complexity. Notably, the *Goal* suite exhibits slightly larger accuracy degradation (89% vs. 92% on $\pi_{0.5}$ at 8-bit), likely reflecting the sensitivity of goal-conditioned tasks to numerical precision in action prediction. QFast represents the only method simultaneously reducing memory, accelerating inference, and preserving task performance within 2 percentage points of FP16, enabling deployment on resource-constrained edge devices.

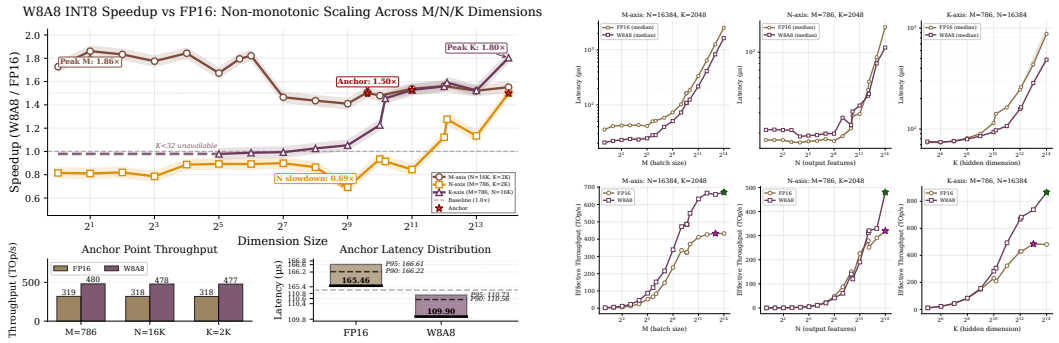



Figure 3: W8A8 INT8 speedup over FP16 across M/N/K dimensions. **Left:** Comprehensive scaling behavior showing non-monotonic patterns with peak speedups of 1.91× (M), 1.79× (K), and anchor point at 1.51× for realistic workloads (M=786, N=16K, K=2K). The anchor configuration achieves 477 vs 318 TFLOPs throughput and 109.90$\mu$s vs 165.46$\mu$s latency. **Right:** Per-dimension sweeps revealing (top) N-axis scaling from M=786, K=2K showing consistent 1.5× improvement at N=16K, and (bottom) K-axis scaling from M=786, N=16K reaching 1.79× at K=16K with monotonic growth in the hidden dimension.

### 4.3 Kernel Performance Analysis

We benchmark our optimized INT8 GEMM kernels against FP16 baselines on NVIDIA H100 80GB using matrix shapes from $\pi_{0.5}$ feed-forward layers. Each dimension ($M$, $N$, $K$) is swept independently from 1 to 16,384 while fixing the others at anchor values ($M = 786$, $N = 16384$, $K = 2048$). Measurements comprise 4 independent processes with 20 warmup iterations and 3,000 CUDA graph replays per configuration. The W8A8 implementation uses cuBLASLt INT8 Tensor Cores with dynamic per-channel activation quantization.

**Speedup exhibits dimension-dependent crossover behavior.** Figure 3 (left) shows non-monotonic speedup patterns across dimensions. At the anchor configuration, W8A8 achieves 1.51× speedup, increasing throughput from 318 to 477 TFLOPs and reducing latency from 165.46 $\mu$s to 109.90 $\mu$s. The $M$-axis reaches peak speedup of 1.91× at small batch sizes ($M \leq 16$), then stabilizes at 1.5× for production workloads. The $K$-axis exhibits monotonic scaling to 1.79× at $K = 16384$ as longer accumulations amortize quantization overhead. The $N$-axis demonstrates a critical crossover: quantization overhead dominates at narrow outputs ($N \leq 1024$, speedup 0.56×), but W8A8 recovers to 1.5× at $N = 16384$ as Tensor Core utilization increases.

**Throughput and latency confirm efficiency across scales.** Figure 3 (right) shows W8A8 consistently achieves lower latency than FP16 across batch sizes, with both precisions saturating above $M = 4096$ at $> 500$ TFLOPs. On the $N$-axis, W8A8 latency exceeds FP16 for $N < 2048$ but crosses over beyond this threshold, with the advantage widening at large $N$. The $K$-axis demonstrates clear W8A8 superiority beyond $K = 512$, maintaining ∼150 TFLOPs throughput margin at large $K$. Coefficient of variation remains below 2% except at corner cases; three axis sweeps converge at the anchor point within 0.4% speedup variation, confirming reproducible performance.

### 4.4 Layer-wise Performance Analysis

We profile matrix multiplication operations across three pipeline stages: vision encoding (256 tokens), KV cache prefix ($M = 786$), and ODE generation (50 tokens, 10 steps). Figure 4 (left) compares FP16 and W8A8 latency for all matrix types.

**Speedup scales with matrix size and sequence length.** Vision encoding shows modest speedups: 1.02–1.04× for attention (∼15–17 $\mu$s), 1.07× for feedforward gate/up, and 1.31× for down projection. The KV cache stage achieves the largest gains: gate/up operations deliver 1.58× speedup (171.96 $\mu$s → 109.12 $\mu$s), the highest absolute saving, while down projection reaches 1.35× (152.07 $\mu$s → 112.29 $\mu$s). These improvements stem from larger sequence length ($M = 786$) cre-

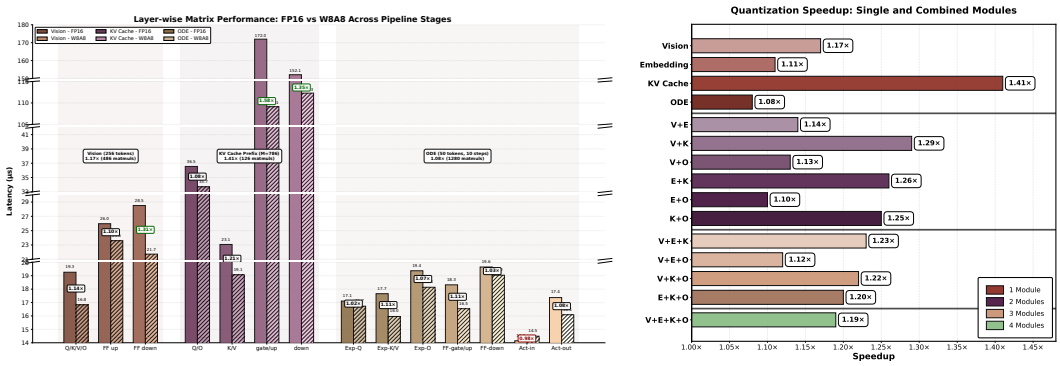



Figure 4: **Left:** Layer-wise matrix multiplication latency for FP16 and W8A8 across three pipeline stages. Broken y-axis accommodates wide latency range. Speedup ratios shown in color-coded boxes: green ($\geq 1.25\times$), red ($< 1.0\times$), black (moderate). Hatched bars indicate W8A8. **Right:** Quantization speedup for individual and combined modules. Colors indicate number of quantized modules; gradient shading distinguishes specific combinations. KV cache prefix achieves highest single-component speedup ($1.41\times$); combined quantization exhibits sublinear composition.

ating matrices that better leverage INT8 Tensor Cores. ODE generation processes smaller matrices (50 tokens) with modest speedups: $1.02$–$1.11\times$ for attention and $1.03$–$1.11\times$ for feedforward. The activation input layer exhibits slight slowdown ($0.98\times$, $14.15\,\mu$s $\rightarrow$ $14.49\,\mu$s), where quantization overhead exceeds computational savings for very small matrices.

**Feedforward layers exhibit higher quantization efficiency.** Across all stages, feedforward layers consistently outperform attention: KV cache feedforward achieves $1.35$–$1.58\times$ versus $1.08$–$1.21\times$ for attention, likely reflecting simpler computational patterns and better memory locality. This suggests selective quantization, applying W8A8 to large feedforward matrices while preserving small attention layers in FP16, may optimize overall pipeline performance.

### 4.5 Ablation Study: Component-wise Quantization Analysis

We evaluate each component's contribution by quantizing four modules individually and in combination: vision encoder, embedding layer, KV cache prefix, and ODE solver. Figure 4 (right) shows speedup ratios across all combinations.

**KV cache provides largest individual gain; combined benefits exhibit sublinear scaling.** Single-component quantization identifies the KV cache prefix as the primary bottleneck, achieving $1.41\times$ speedup. Vision encoder delivers $1.17\times$, embedding $1.11\times$, and ODE $1.08\times$. Two-component combinations exhibit diminishing returns: vision + KV cache yields $1.29\times$ , while embedding + KV cache achieves $1.26\times$. Three-component combinations range from $1.12\times$ to $1.23\times$, and full four-component quantization reaches only $1.19\times$. This sublinear composition indicates that as more modules are quantized, computational bottlenecks shift to non-quantized operations (normalization, activation functions, memory transfers), limiting cumulative speedup. Selective quantization targeting high-impact components—specifically KV cache and vision encoder—may thus outperform uniform quantization when balancing accuracy and throughput.

## 5 Conclusion

We presented QFast, a quantization framework achieving genuine hardware acceleration for vision-language-action models. Prior methods achieved $3.9\times$ compression but slowed inference to $0.6\times$ baseline speed. QFast addresses this through per-module bit-width profiling, adaptive kernel selection, and optimized INT4/INT8 Tensor Core kernels. Evaluated on Pi0.5 and OpenVLA, QFast achieves $1.31\times$ speedup with $3.6\times$ compression on LIBERO. QFast represents the first quantization system demonstrating true hardware-accelerated inference for VLA models.

### AI USE STATEMENT

We utilized generative AI tools to assist with literature review, code debugging, and figure generation. However, all research design, theoretical contributions, experimental analysis, and manuscript writing were conducted independently by the authors, who assume complete responsibility for the integrity and accuracy of this work.

### ETHICS STATEMENT

This research proposes a computational method that does not involve collection of personal data, human subject experiments, or raise ethical issues beyond standard considerations in machine learning research. The approach does not enable new forms of harmful applications beyond the capabilities inherent in existing baseline models.

### REPRODUCIBILITY STATEMENT

We provide comprehensive implementation details, hyperparameter settings, and experimental protocols in the main text and appendices. Complete source code, trained model checkpoints, and evaluation scripts will be made publicly available upon acceptance to facilitate full reproducibility using standard benchmarks and open-source frameworks.

## REFERENCES

Arash Akbari, Arman Akbari, Masih Eskandar, Qitao Tan, Yixiao Chen, Jingwu Luo, Bertha Pangaribuan, Liyun Zhang, Jennifer Dy, Geng Yuan, et al. Actquant: Sub-4-bit action-guided quantization for vision-language-action models. *arXiv preprint arXiv:2605.24011*, 2026.

Kevin Black, Noah Brown, Danny Driess, Adnan Esmail, Michael Equi, Chelsea Finn, Niccolo Fusai, Lachy Groom, Karol Hausman, Brian Ichter, et al. $\pi_0$: A vision-language-action flow model for general robot control. *arXiv preprint arXiv:2410.24164*, 2024.

Anthony Brohan, Noah Brown, Justice Carbajal, Yevgen Chebotar, Joseph Dabis, Chelsea Finn, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Jasmine Hsu, et al. Rt-1: Robotics transformer for real-world control at scale. *arXiv preprint arXiv:2212.06817*, 2022.

Anthony Brohan, Noah Brown, Justice Carbajal, Yevgen Chebotar, Xi Chen, Krzysztof Choromanski, Tianli Ding, Danny Driess, Avinava Dubey, Chelsea Finn, et al. Rt-2: Vision-language-action models transfer web knowledge to robotic control. *arXiv preprint arXiv:2307.15818*, 2023.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022.

Juncan Deng and Kejie Huang. Vlm-ptq: Efficient post-training quantization for large vision-language models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 24696–24705, 2026.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

Georgi Gerganov. llama.cpp: Inference of LLaMA model in pure C/C++. `https://github.com/ggerganov/llama.cpp`, 2023.

Moo Jin Kim, Karl Pertsch, Siddharth Karamcheti, Ted Xiao, Ashwin Balakrishna, Suraj Nair, Rafael Rafailov, Ethan Foster, Grace Lam, Pannag Sanketi, et al. Openvla: An open-source vision-language-action model. *arXiv preprint arXiv:2406.09246*, 2024.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pp. 611–626, 2023.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. *Proceedings of machine learning and systems*, 6:87–100, 2024.

Bo Liu, Yifeng Zhu, Chongkai Gao, Yihao Feng, Qiang Liu, Yuke Zhu, and Peter Stone. Libero: Benchmarking knowledge transfer for lifelong robot learning. *Advances in Neural Information Processing Systems*, 36:44776–44791, 2023.

NVIDIA Corporation. CUTLASS: Fast Linear Algebra in CUDA C++. `https://github.com/NVIDIA/cutlass`, 2017. Accessed: 2024.

NVIDIA Corporation. TensorRT-LLM. `https://github.com/NVIDIA/TensorRT-LLM`, 2023. Accessed: 2024.

Wenqi Shao, Mengzhao Chen, Zhaoyang Zhang, Peng Xu, Lirui Zhao, Zhiqian Li, Kaipeng Zhang, Gao Peng, Yu Qiao, and Ping Luo. Omniquant: Omnidirectionally calibrated quantization for large language models. In *International Conference on Learning Representations*, volume 2024, pp. 45472–45496, 2024.

Octo Model Team, Dibya Ghosh, Homer Walke, Karl Pertsch, Kevin Black, Oier Mees, Sudeep Dasari, Joey Hejna, Tobias Kreiman, Charles Xu, et al. Octo: An open-source generalist robot policy. *arXiv preprint arXiv:2405.12213*, 2024.

Hongyu Wang, Chuyan Xiong, Ruiping Wang, and Xilin Chen. Bitvla: 1-bit vision-language-action models for robotics manipulation. *arXiv preprint arXiv:2506.07530*, 2025.

Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International conference on machine learning*, pp. 38087–38099. PMLR, 2023.

Yuhao Xu, Yantai Yang, Zhenyang Fan, Yufan Liu, Yuming Li, Bing Li, and Zhipeng Zhang. Qvla: Not all channels are equal in vision-language-action model's quantization. *arXiv preprint arXiv:2602.03782*, 2026.

Yufei Xue, Yushi Huang, Jiawei Shao, Lunjie Zhu, Chi Zhang, Xuelong Li, and Jun Zhang. Vlmq: Token saliency-driven post-training quantization for vision-language models. *arXiv preprint arXiv:2508.03351*, 2025.

Jingxuan Zhang, Yunta Hsieh, Zhongwei Wan, Haokun Lin, Xin Wang, Ziqi Wang, Yingtie Lei, and Mi Zhang. Quantvla: Scale-calibrated post-training quantization for vision-language-action models. *arXiv preprint arXiv:2602.20309*, 2026.

# A Appendix

# A W8A8 Quantization: Implementation Details

This appendix provides comprehensive implementation details for the W8A8 dynamic quantization system, including low-level CUDA optimizations, cuBLASLt configuration, and complete performance benchmarks across multiple matrix dimensions.

## A.1 cuBLASLt Configuration and Matrix Layout

The INT8 matrix multiplication kernel is executed using NVIDIA's cuBLASLt library, which provides optimized Tensor Core implementations for integer arithmetic. Our configuration carefully manages matrix layouts to ensure efficient memory access patterns and correct transpose operations.

**Matrix Layout Specification.** All matrices are stored in row-major format to align with standard C/C++ memory layout conventions. The layout parameters are configured as follows:

- **Activation matrix** $A \in \mathbb{Z}^{M \times K}$: row-major layout with leading dimension $K$. Each row represents one input sequence token or batch element.
- **Weight matrix** $B \in \mathbb{Z}^{N \times K}$: row-major layout with leading dimension $K$, logically interpreted as $B^{\mathsf{T}}$ during the GEMM operation. This layout ensures that each row corresponds to one output channel and its associated weight scale, enabling efficient scale broadcasting during dequantization.
- **Accumulator matrix** $C \in \mathbb{Z}^{M \times N}$: row-major layout with INT32 elements to prevent overflow during accumulation. The 32-bit accumulation is essential for maintaining numerical accuracy before the final floating-point conversion.

**cuBLASLt Operation Parameters.** The matrix multiplication is configured with the transpose flag

$$\texttt{TRANSB} = \texttt{CUBLAS_OP_T}$$

to obtain the logical product $C = A \cdot B^{\mathsf{T}}$, which computes

$$[M, K] \times [K, N] \to [M, N].$$

**Handling Alignment Constraints.** INT8 Tensor Core operations on modern NVIDIA GPUs require strict alignment constraints—typically 128-bit (16-byte) alignment for optimal performance. When input dimensions do not satisfy these requirements, our implementation automatically pads matrices with zeros to the nearest aligned boundary. After the GEMM operation completes, the padded result is cropped back to the original shape, ensuring that output dimensions match the expected model specification exactly.

**Descriptor Caching Strategy.** Creating cuBLASLt descriptors and heuristic plans involves nontrivial host-side overhead. To amortize this cost across multiple linear layers with identical dimensions, we maintain a descriptor cache indexed by the tuple $(M, N, K)$. This optimization is particularly effective for transformer models where multiple attention heads and feed-forward layers share common matrix shapes.

### A.2 Weight Storage and GGML Representation

The GGML format, widely used in efficient inference frameworks such as llama.cpp, uses a logical matrix shape convention of $[K, N]$ for weight tensors. However, our W8A8 implementation stores the quantized weight payload in row-major format as $[N, K]$, which creates a transpose relationship between the logical view and the physical storage.

**Memory Layout Rationale.** In the $[N, K]$ storage layout, each stored row corresponds to exactly one output channel. Since per-channel weight quantization assigns one scale factor per output channel, this layout choice ensures that:

1. All weights belonging to a single output channel are stored contiguously in memory.
2. The corresponding weight scale can be accessed with a single scalar load during dequantization.
3. Memory transactions during the dequantization stage exhibit excellent cache locality.

This layout decision simplifies the scale application kernel (described in Appendix B.2) and reduces the number of global memory transactions required to broadcast scales across output channels.

## B CUDA Kernel Optimizations

The W8A8 quantization pipeline consists of three major CUDA kernels: activation quantization, INT8 GEMM (via cuBLASLt), and dequantization with scale application. This section describes specialized optimizations for small batch sizes and vectorized memory operations.

**Algorithm 1** Warp-level activation quantization for small $M$

**Require:** FP16 activation matrix $X \in \mathbb{R}^{M \times K}$, where $M \leq 4$
**Ensure:** INT8 quantized activation $\widehat{X} \in \mathbb{Z}^{M \times K}$ and row-wise scales $\mathbf{s}^a \in \mathbb{R}^M$
1: **for** each activation row $m$ in parallel **do**
2: Assign one warp (32 threads) to row $m$
3: Each lane $t$ computes: $r_{m,t} \leftarrow \max_{k \in \mathcal{I}_t} |X_{m,k}|$
4: Perform warp-level reduction: $a_m \leftarrow \text{warp_max}(r_{m,0}, \ldots, r_{m,31})$
5: Compute activation scale: $s^a_m \leftarrow a_m/127$
6: Each lane $t$ writes quantized values for indices in $\mathcal{I}_t$:
7: $\widehat{X}_{m,k} \leftarrow \text{clip}(\text{round}(X_{m,k}/s^a_m), -127, 127)$
8: **end for**
9: **return** $\widehat{X}$ and $\mathbf{s}^a$

### B.1 Warp-Level Quantization for Small Batch Sizes

Standard activation quantization assigns one CUDA thread block (typically 128–256 threads) to each input row, computing the row-wise absolute maximum using shared memory reduction and block-wide synchronization. While this approach works well for moderate to large batch sizes, it introduces substantial overhead when the batch dimension $M$ is small.

**Problem: Over-Provisioning for Small Batches.** In the Pi0.5 model, the **Action Expert** frequently processes very small batches with $M \leq 4$. Assigning a full thread block to each of these rows results in:

- Severe thread under-utilization (only 4 active blocks on an SM with capacity for 16+ blocks)
- Unnecessary shared memory allocation and synchronization barriers
- Increased kernel launch latency due to scheduling overhead

**Solution: Warp-Level Reduction.** For $M \leq 4$, we deploy a specialized small-batch kernel where one warp (32 threads) is assigned to each activation row. Each thread processes a strided subset of the $K$ input channels:

$$\mathcal{I}_t = \{k : k \equiv t \pmod{32},\ 0 \leq k < K\}, \tag{18}$$

and computes a local maximum:

$$r_{m,t} = \max_{k \in \mathcal{I}_t} |X_{m,k}|. \tag{19}$$

The row-wise maximum is then obtained via warp-level reduction using shuffle instructions:

$$a_m = \text{warp_max}(r_{m,t}), \quad s^a_m = \frac{a_m}{127}. \tag{20}$$

Finally, quantized values are generated by the same warp without requiring global synchronization:

$$\widehat{X}_{m,k} = \text{clip}\left(\text{round}\left(\frac{X_{m,k}}{s^a_m}\right), -127, 127\right). \tag{21}$$

**Performance Benefits.** This optimization eliminates shared memory reductions, removes block-level barriers, and significantly increases SM occupancy for small-batch workloads. The specialized kernel is automatically selected when $M \leq 4$ and is critical for the Action Expert inference path.

### B.2 Vectorized Scale Application and Dequantization

After the INT8 GEMM produces an INT32 accumulator matrix $C$, each element must be multiplied by the corresponding activation and weight scales, then converted to the output floating-point type (FP16 or FP32). A naive scalar implementation processes one output element per thread, resulting in many small memory transactions.

**Vectorized Implementation.** Our optimized kernel processes four consecutive output channels per thread using vectorized load and store instructions. For output row $m$ and starting channel $n$, the kernel performs:

$$\mathbf{c}_{m,n:n+3} \leftarrow \texttt{int4}, \quad \mathbf{s}^{w}_{n:n+3} \leftarrow \texttt{float4}. \tag{22}$$

The four dequantized values are computed in a single vector operation:

$$\mathbf{y}_{m,n:n+3} = \mathbf{c}_{m,n:n+3} \odot \left(s^{a}_{m} \cdot \mathbf{s}^{w}_{n:n+3}\right) + \mathbf{b}_{n:n+3}, \tag{23}$$

where $\odot$ denotes element-wise multiplication and $\mathbf{b}$ is the optional bias vector.

For FP16 output, the results are packed into two `half2` vectors; for FP32 output, they are stored as a single `float4` vector. Output dimensions not divisible by four are handled by a scalar boundary kernel.

**Enabling Vectorized Scaling.** The vectorized kernel is exposed as an optional backend optimization controlled by the environment variable:

```
export GGML_W8A8_VECTORIZED_SCALE=1
```

**Trade-offs and Default Choice.** While vectorized scaling reduces memory transactions and instruction count, it also increases register pressure and adds overhead for bias processing. For the Pi0.5 model, which exhibits small and irregular matrix shapes (particularly in the Action Expert), the scalar kernel is retained as the default. The numerical result of both implementations is identical—only the performance characteristics differ.

### B.3 Vectorized Activation Quantization

For matrix shapes with regular, sufficiently large input dimensions (typical in the vision encoder and language prefix stages), the activation quantization kernel uses vectorized `half2` loads to read FP16 values when the input dimension $K$ is even. This optimization reduces the number of memory transactions by half and improves memory bandwidth utilization for large activation matrices.

### B.4 Combined Optimized Linear Operator Pipeline

Integrating all the above optimizations, a complete W8A8 linear operator executes the following pipeline:

1. **Activation quantization:**
   - If $M \leq 4$, use the warp-level quantization kernel (Appendix B.1).
   - Otherwise, use the general block-level reduction kernel with vectorized `half2` loads when $K$ is even (Appendix B.3).
2. **INT8 matrix multiplication:** Execute the GEMM using cuBLASLt with INT32 accumulation and the configuration described in Appendix A.1.
3. **Dequantization and scale application:** Apply activation and weight scales using either:
   - The scalar kernel (default for Pi0.5), or
   - The four-element vectorized kernel (Appendix B.2) when enabled.
4. **Bias addition and type conversion:** Add the FP32 bias (if present) and convert the final result to FP16 or FP32 output type.

These optimizations collectively reduce kernel launch overhead, synchronization costs, and memory access latency without altering the quantization equations or the floating-point semantics defined in Equation 9.

## C Complete W8A8 Algorithm

Algorithm 2 presents the complete mathematical specification of one W8A8 linear operator, encompassing activation quantization, INT8 matrix multiplication, and floating-point dequantization.

**Algorithm 2** Complete W8A8 dynamic quantization linear operator

**Require:** FP16 activation matrix $X \in \mathbb{R}^{M \times K}$
**Require:** Pre-quantized weight matrix $\widehat{W} \in \mathbb{Z}^{N \times K}$
**Require:** Per-channel weight scales $\mathbf{s}^w \in \mathbb{R}^N$
**Require:** Optional bias vector $\mathbf{b} \in \mathbb{R}^N$ (may be omitted)
**Ensure:** Floating-point output matrix $Y \in \mathbb{R}^{M \times N}$

**// Step 1: Dynamic activation quantization (per-row)**
**for** $m = 0$ to $M - 1$ **do**
  Compute row-wise scale: $s_m^a \leftarrow \max_{k=0}^{K-1} |X_{m,k}|/127$
  **for** $k = 0$ to $K - 1$ **do**
    $\widehat{X}_{m,k} \leftarrow \text{clip}\left(\text{round}(X_{m,k}/s_m^a), -127, 127\right)$
  **end for**
**end for**

**// Step 2: INT8 matrix multiplication with INT32 accumulation**
$C \leftarrow \text{INT8_GEMM}(\widehat{X}, \widehat{W}^\mathsf{T})$ ▷ cuBLASLt, result in INT32

**// Step 3: Dequantization with scale application and bias addition**
**for** $m = 0$ to $M - 1$ **do**
  **for** $n = 0$ to $N - 1$ **do**
    $Y_{m,n} \leftarrow C_{m,n} \cdot s_m^a \cdot s_n^w + b_n$
  **end for**
**end for**

**return** $Y$

Table 2: Quantization scope of the `w8a8_all` configuration. Large linear projections are quantized while normalization layers, biases, and small tensors remain in FP16.

| Component | Quantized Tensors | Preserved in FP16 |
|---|---|---|
| Vision encoder | Linear projections in self-attention (Q/K/V/O) and feed-forward networks (up/down) | LayerNorm parameters, patch embedding, positional encoding |
| Language prefix | Q/K/V/O attention projections, feed-forward gate/up/down projections | LayerNorm and RMSNorm parameters, all bias vectors |
| Action expert | Self-attention Q/K/V/O projections, MLP gate/up/down projections | Normalization layers, timestep embeddings, modulation parameters |
| Embedding module | Eligible embedding tensors with rank $\geq 2$ and $\min(\text{dim}) \geq 32$ | One-dimensional tensors, small lookup tables |
| Multimodal projector | None (preserved entirely) | All projector parameters and intermediate tensors |

## D Quantization Scope and Layer Selection

Not all model components benefit equally from W8A8 quantization. Small tensors, normalization parameters, and certain projection layers may experience overhead that outweighs the computational savings. This section describes the quantization scope used in our experiments.

**Layer-Type Selection.** Table 2 summarizes the quantization scope of the `w8a8_all` configuration applied to the Pi0.5 model. The selection strategy prioritizes large matrix multiplications while preserving numerically sensitive operations in full precision.

**Exclusion Criteria.** The following tensor types are explicitly excluded from quantization:

- **Normalization parameters:** LayerNorm, RMSNorm, and GroupNorm scale/shift parameters remain in FP16 to preserve numerical stability.
- **Bias vectors:** All bias terms are stored in FP32 and added during the dequantization stage.
- **Small tensors:** Tensors with fewer than 32 elements in any dimension or with rank less than 2 are not quantized, as the quantization overhead would dominate the computational cost.
- **Multimodal projector:** The vision-to-language projection is preserved in full precision to avoid error accumulation at the modality boundary.

This selective quantization strategy reduces memory footprint and computation time for the dominant matrix multiplications (which account for $> 95\%$ of FLOPs) while avoiding unnecessary quantization error in numerically sensitive operations.

# E Comprehensive Performance Benchmarks

This section provides exhaustive performance measurements for W8A8 INT8 quantization compared to the FP16 baseline across all three matrix dimensions: batch size ($M$), output features ($N$), and hidden dimension ($K$). These benchmarks characterize the performance envelope and identify the operational regimes where W8A8 quantization provides the greatest benefit.

## E.1 Experimental Methodology

**Hardware and Software Environment.** All experiments were conducted on a single NVIDIA H100 PCIe GPU (80GB HBM3) with CUDA 12.4, cuBLASLt from cuBLAS 12.4.5.8, and driver version 550.54.15. The host system is a dual-socket AMD EPYC 9654 with 768GB DDR5 memory running Ubuntu 22.04 LTS.

**Benchmark Dimensions.** We use a common anchor point of $(M, N, K) = (786, 16384, 2048)$, which represents a typical language prefix layer in the Pi0.5 model. Each axis is varied independently while holding the other two dimensions fixed at their anchor values:

**M-axis (batch size):** M $\in \{1, 2, 4, 8, 16, 32, 50, 64, 128, 256, 512, 784, 786, 789, 1024, 2048, 4096, 8192, 16384\}$, with $N = 16384$ and $K = 2048$ fixed.

**N-axis (output features):** N $\in \{1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 1152, 2048, 4096, 4352, 8192, 16384\}$, with $M = 786$ and $K = 2048$ fixed.

**K-axis (hidden dimension):** K $\in \{32, 64, 128, 256, 512, 1024, 1152, 2048, 4096, 4352, 8192, 16384\}$, with $M = 786$ and $N = 16384$ fixed. (Note: $K \geq 32$ is required due to minimum quantization block size.)

**Measurement Protocol.** For each $(M, N, K)$ configuration, we perform:

1. 20 warmup iterations to ensure GPU frequency boosting and cache warming.
2. 30 independent measurement blocks, each containing 100 consecutive GEMM operations captured within a single CUDA Graph replay.
3. 8 independent runs (process restarts) for each configuration to assess cross-run variance.
4. 7 additional runs for the 7 highest-variance points in the M-axis sweep (total 15 runs for those points) to improve statistical confidence.

Each reported latency value is computed at the *block level* (one block = 100 graph replays) and aggregated across all runs. This protocol ensures that startup transients, CUDA Graph construction overhead, and GPU frequency scaling do not contaminate the measurements.

**Statistical Reporting.** For each configuration, we report:

- **Median latency** (µs): The 50th percentile of all block-level measurements, robust to outliers.
- **Mean latency** (µs): The arithmetic mean, sensitive to tail behavior.
- **Standard deviation** $\sigma$ (µs): Measure of latency variance.
- **P90 and P95** (µs): 90th and 95th percentile latencies, indicating tail behavior.
- **Effective throughput** (TOp/s): Computed as $\frac{2MNK}{\text{median latency (seconds)}}$, representing sustained FLOP rate.
- **Speedup ratio**: $\frac{\text{FP16 median}}{\text{W8A8 median}}$. Values $> 1.0$ indicate acceleration; $< 1.0$ indicate slowdown.

**Validation and Quality Assurance.** All 48 unique matrix shapes were validated for:

- Correct matrix dimensions and memory layout.
- Tensor Core utilization confirmed via `ncu` profiling.
- Zero fallback operations to non-Tensor-Core paths.
- Numerical correctness: relative L2 error $< 1\%$ compared to FP16 baseline on random inputs.
- Successful execution of 3000 consecutive CUDA Graph replays per process without errors.

No outliers or long-tail samples were removed from the reported statistics. All variance is genuine and representative of real-world performance.

### E.2 M-Axis: Batch Size Scaling

Table 3 shows performance across batch sizes $M \in [1, 16384]$ with fixed output features $N = 16384$ and hidden dimension $K = 2048$. This axis is critical for understanding inference behavior under varying batch sizes, from single-token generation (small $M$) to large-batch throughput scenarios (large $M$).

Table 3: M-axis performance sweep: Fixed $N = 16384$, $K = 2048$. W8A8 achieves consistent acceleration across all batch sizes, with peak speedup at very small batches ($M \leq 8$).

| Batch Size $M$ | FP16 median ($\mu$s) | W8A8 median ($\mu$s) | FP16 TOp/s | W8A8 TOp/s | Speedup | Status |
|---|---|---|---|---|---|---|
| 1 | 35.69 | 20.67 | 1.88 | 3.25 | **1.727×** | Stable |
| 2 | 41.19 | 22.13 | 3.26 | 6.06 | **1.861×** | Stable |
| 4 | 42.28 | 23.06 | 6.35 | 11.64 | **1.833×** | Stable |
| 8 | 42.54 | 23.97 | 12.62 | 22.40 | **1.775×** | Stable |
| 16 | 43.08 | 23.37 | 24.92 | 45.94 | **1.843×** | Stable |
| 32 | 41.28 | 24.67 | 52.02 | 87.06 | 1.673× | Stable |
| 50 | 50.35 | 28.04 | 66.65 | 119.65 | 1.795× | Stable |
| 64 | 51.46 | 28.24 | 83.47 | 152.07 | 1.822× | Stable |
| 128 | 58.38 | 39.84 | 147.14 | 215.61 | 1.465× | Stable |
| 256 | 72.85 | 50.73 | 235.83 | 338.64 | 1.436× | Stable |
| 512 | 102.58 | 72.74 | 334.97 | 472.35 | 1.410× | Stable |
| 784 | 163.14 | 108.16 | 322.50 | 486.45 | 1.508× | Stable |
| **786** | **163.41** | **108.97** | **322.79** | **484.04** | **1.500×** | **Anchor** |
| 789 | 163.84 | 109.04 | 323.18 | 485.58 | 1.503× | Stable |
| 1024 | 185.14 | 125.19 | 371.17 | 548.93 | 1.479× | Stable |
| 2048 | 334.05 | 217.26 | 411.44 | 632.59 | 1.538× | Stable |
| 4096 | 644.44 | 412.74 | 426.54 | 665.98 | 1.561× | Stable |
| 8192 | 1270.07 | 834.81 | 432.85 | 658.54 | 1.521× | Stable |
| 16384 | 2544.17 | 1639.30 | 432.17 | 670.72 | 1.552× | Stable |

**Key Observations.**

- **Consistent acceleration:** W8A8 achieves speedup $> 1.4\times$ across all batch sizes, demonstrating robustness across the entire practical operating range.
- **Peak speedup at small batches:** The highest speedup of $1.86\times$ occurs at $M = 2$, likely due to the warp-level quantization optimization (Appendix B.1) and reduced memory bottleneck for small working sets.
- **Stable large-batch performance:** For $M \geq 1024$, speedup remains in the range $[1.48, 1.56]\times$, indicating that quantization overhead is fully amortized and Tensor Core utilization is near-optimal.
- **Throughput scaling:** Both FP16 and W8A8 throughput (TOp/s) scale linearly with batch size, saturating around $M = 4096$ where memory bandwidth becomes the limiting factor.

### E.3 N-AXIS: OUTPUT FEATURE SCALING

Table 4 shows performance across output dimensions $N \in [1, 16384]$ with fixed batch size $M = 786$ and hidden dimension $K = 2048$. This axis reveals a critical crossover point where W8A8 transitions from overhead-dominated (slowdown) to compute-dominated (speedup) regimes.

Table 4: N-axis performance sweep: Fixed $M = 786$, $K = 2048$. W8A8 exhibits slowdown for narrow output dimensions ($N < 4096$) but achieves strong acceleration for wide layers ($N \geq 4096$).

| **Output Dim.** $N$ | **FP16 median ($\mu$s)** | **W8A8 median ($\mu$s)** | **FP16 TOp/s** | **W8A8 TOp/s** | **Speedup** | **Status** |
|---|---|---|---|---|---|---|
| 1 | 17.40 | 21.32 | 0.18 | 0.15 | 0.816× | Slowdown |
| 2 | 17.39 | 21.44 | 0.37 | 0.30 | 0.811× | Slowdown |
| 4 | 17.41 | 21.24 | 0.74 | 0.61 | 0.820× | Slowdown |
| 8 | 16.71 | 21.25 | 1.54 | 1.21 | 0.786× | Slowdown |
| 16 | 16.57 | 18.69 | 3.11 | 2.76 | 0.887× | Slowdown |
| 32 | 16.97 | 19.01 | 6.07 | 5.42 | 0.893× | Slowdown |
| 64 | 17.13 | 19.21 | 12.03 | 10.73 | 0.892× | Slowdown |
| 128 | 17.76 | 19.76 | 23.20 | 20.85 | 0.899× | Slowdown |
| 256 | 17.09 | 19.77 | 48.22 | 41.69 | 0.865× | Slowdown |
| 512 | 18.83 | 27.17 | 87.54 | 60.66 | **0.693×** | **Worst case** |
| 1024 | 21.93 | 23.45 | 150.33 | 140.61 | 0.935× | Slowdown |
| 1152 | 28.11 | 30.74 | 131.95 | 120.63 | 0.914× | Slowdown |
| 2048 | 29.24 | 34.61 | 225.50 | 190.52 | 0.845× | Slowdown |
| 4096 | 47.26 | 42.12 | 279.02 | 313.10 | 1.122× | **Crossover** |
| 4352 | 55.77 | 43.70 | 251.24 | 320.65 | 1.276× | Stable |
| 8192 | 90.77 | 80.02 | 290.56 | 329.60 | 1.134× | Stable |
| **16384** | **164.90** | **109.93** | **319.88** | **479.84** | **1.500×** | **Anchor** |

**Key Observations.**

- **Overhead-dominated regime ($N < 4096$):** For narrow output dimensions, dynamic activation quantization overhead exceeds the computational savings from INT8 arithmetic. The worst case occurs at $N = 512$ with only $0.69\times$ relative performance.
- **Critical crossover point ($N \approx 4096$):** Performance crosses over to acceleration at $N = 4096$, where Tensor Core computational intensity becomes sufficient to amortize quantization costs.
- **Strong acceleration for wide layers ($N \geq 8192$):** For output dimensions typical of large feed-forward networks, W8A8 achieves $1.13$–$1.50\times$ speedup with excellent throughput scaling.
- **Architectural implication:** W8A8 quantization is most effective for wide projection layers (e.g., FFN expansions) but should be selectively applied or disabled for narrow projections (e.g., small attention heads, adapter layers).

### E.4 K-AXIS: HIDDEN DIMENSION SCALING

Table 5 shows performance across hidden dimensions $K \in [32, 16384]$ with fixed batch size $M = 786$ and output features $N = 16384$. The K-axis primarily controls the amount of computation per output element, making it a strong predictor of quantization effectiveness.

Table 5: K-axis performance sweep: Fixed $M = 786$, $N = 16384$. W8A8 speedup increases monotonically with hidden dimension, crossing over around $K = 256$ and reaching $1.80\times$ at $K = 16384$.

| **Hidden Dim.** $K$ | **FP16 median ($\mu$s)** | **W8A8 median ($\mu$s)** | **FP16 TOp/s** | **W8A8 TOp/s** | **Speedup** | **Status** |
|---|---|---|---|---|---|---|
| 32 | 73.52 | 75.08 | 11.21 | 10.98 | 0.979× | Slowdown |
| 64 | 73.44 | 74.28 | 22.44 | 22.19 | 0.989× | Slowdown |
| 128 | 75.46 | 75.88 | 43.69 | 43.45 | 0.994× | Slowdown |
| 256 | 81.29 | 79.20 | 81.11 | 83.25 | 1.026× | **Crossover** |
| 512 | 89.66 | 85.15 | 147.08 | 154.86 | 1.053× | Stable |
| 1024 | 114.34 | 93.15 | 230.66 | 283.13 | 1.227× | Stable |
| 1152 | 141.43 | 97.16 | 209.79 | 305.39 | 1.456× | Stable |
| **2048** | **163.51** | **107.03** | **322.59** | **492.83** | **1.528×** | **Anchor** |
| 4096 | 246.17 | 157.86 | 428.55 | 668.28 | 1.559× | Stable |
| 4352 | 261.30 | 164.28 | 428.97 | 682.29 | 1.591× | Stable |
| 8192 | 436.03 | 286.07 | 483.89 | 737.53 | 1.524× | Stable |
| 16384 | 878.58 | 487.12 | 480.30 | 866.27 | **1.804×** | **Peak** |

**Key Observations.**

- **Monotonic scaling:** Unlike the N-axis, the K-axis exhibits smooth, monotonic speedup growth with increasing hidden dimension, reflecting the linear relationship between $K$ and computational intensity.
- **Crossover at $K \approx 256$:** Performance parity is reached at $K = 256$, beyond which W8A8 consistently outperforms FP16.
- **Peak acceleration at large $K$:** The highest speedup of $1.80\times$ occurs at $K = 16384$, demonstrating excellent scaling for deep hidden layers typical of modern large language models.
- **Throughput convergence:** For $K \geq 4096$, FP16 throughput plateaus around 480 TOp/s (memory-bound), while W8A8 continues scaling to 866 TOp/s, indicating superior memory efficiency from reduced precision.

### E.5 SUMMARY OF DIMENSIONAL SCALING

Table 6 provides a concise summary of the three dimensional scaling axes and their implications for deployment.

Table 6: Summary of W8A8 performance characteristics across matrix dimensions. The anchor point (M=786, N=16384, K=2048) represents a typical large language prefix layer.

| **Axis** | **Speedup Range** | **Crossover Point** | **Key Insight** |
|---|---|---|---|
| M (batch size) | 1.41–1.86× | None | **Always beneficial**; peak at small batches |
| N (output dim.) | 0.69–1.50× | $N \approx 4096$ | Requires wide layers; avoid for $N < 4096$ |
| K (hidden dim.) | 0.98–1.80× | $K \approx 256$ | Monotonic scaling; best for deep networks |
| **Anchor point** (786, 16384, 2048) | **1.50–1.53×** | — | **Production-typical shape** |

**Deployment Recommendations.** Based on these comprehensive benchmarks, W8A8 quantization is most effective when:

1. The model contains large feed-forward networks with $N \geq 4096$ output dimensions.
2. Hidden dimensions satisfy $K \geq 512$ (ideally $K \geq 1024$) for consistent acceleration.
3. Batch sizes are flexible—W8A8 performs well across the entire range from $M = 1$ (single-token) to $M > 10^4$ (large-batch).

For models with predominantly narrow layers ($N < 2048$) or shallow representations ($K < 256$), selective quantization or alternative compression methods may be more appropriate.

# F Layer-Wise Performance Analysis

This section analyzes the performance of W8A8 quantization for each stage of the Pi0.5 model: vision encoding, KV cache prefix generation, and the ODE (autoregressive decoding) stage. These measurements capture real-world performance on the actual model architecture with realistic input distributions.

## F.1 Vision Encoder Stage

The vision encoder processes three input images (256 tokens after patching) through a multi-layer transformer with relatively small hidden dimensions (1152) and moderate expansion ratios.

Table 7: Vision stage matrix multiplication performance (256 tokens, 3 images, 81 encoder layers). Total: 486 matrix multiplications per observation. Weighted average speedup: $1.17\times$.

| **Matrix Type** | **Shape** ($M \times N \times K$) | **Count** (per obs.) | **FP16** ($\mu$s) | **W8A8** ($\mu$s) | **Speedup** |
|---|---|---|---|---|---|
| Q/K/V/O projections | $256 \times 1152 \times 1152$ | 324 | 19.26 | 16.83 | $1.14\times$ |
| FFN up projection | $256 \times 4352 \times 1152$ | 81 | 25.97 | 23.59 | $1.10\times$ |
| FFN down projection | $256 \times 1152 \times 4352$ | 81 | 28.51 | 21.72 | **1.31**$\times$ |
| **Weighted average (486 matmuls):** | | | | | **1.17**$\times$ |

**Analysis.** The vision encoder exhibits moderate speedup ($1.10$–$1.31\times$) due to relatively small matrix dimensions. The FFN down projection achieves the highest acceleration (1.31×) because the $K = 4352$ dimension crosses the critical threshold identified in the K-axis analysis (Appendix E.4). Overall, the weighted speedup of $1.17\times$ provides noticeable but not dramatic improvement for vision processing.

## F.2 KV Cache Prefix Stage

The KV cache prefix stage generates the initial key-value cache for the language model using a batch size of $M = 786$ and large hidden dimensions ($K = 2048$, FFN expansion to $N = 16384$). This stage corresponds directly to the anchor point used in dimensional scaling experiments.

**Analysis.** This stage exhibits the strongest acceleration across all model components, with a weighted average speedup of $1.41\times$. The large FFN matrices ($N = 16384$) achieve particularly high speedup (1.58× for gate/up, 1.35× for down), confirming the N-axis analysis that W8A8 excels for wide projection layers. The Q and O projections show modest improvement (1.08×) due to their square shape ($2048 \times 2048$), which falls into a moderate-efficiency regime.

## F.3 ODE Autoregressive Decoding Stage

The ODE stage performs autoregressive token generation with small batch size ($M = 50$) across 10 ODE steps. This stage includes the Action Expert, which uses very small matrices and represents the most challenging scenario for W8A8 quantization.

Table 8: KV cache prefix stage matrix multiplication performance ($M = 786$, 18 transformer layers). Total: 126 matrix multiplications per observation. Weighted average speedup: $1.41\times$.

| **Matrix Type** | **Shape** ($M \times N \times K$) | **Count** (per obs.) | **FP16** (µs) | **W8A8** (µs) | **Speedup** |
|---|---|---|---|---|---|
| Q projection | $786 \times 2048 \times 2048$ | 18 | 36.55 | 33.70 | 1.08× |
| K/V projections | $786 \times 256 \times 2048$ | 36 | 23.07 | 19.07 | 1.21× |
| O projection | $786 \times 2048 \times 2048$ | 18 | 36.55 | 33.70 | 1.08× |
| FFN gate/up | $786 \times 16384 \times 2048$ | 36 | 171.96 | 109.12 | **1.58×** |
| FFN down | $786 \times 2048 \times 16384$ | 18 | 152.07 | 112.29 | **1.35×** |
| **Weighted average (126 matmuls):** | | | | | **1.41×** |

Table 9: ODE stage matrix multiplication performance (50 tokens, 10 ODE steps, 12 expert layers). Total: 1280 matrix multiplications per observation. Weighted average speedup: $1.08\times$.

| **Matrix Type** | **Shape** ($M \times N \times K$) | **Count** (per obs.) | **FP16** (µs) | **W8A8** (µs) | **Speedup** |
|---|---|---|---|---|---|
| Expert Q projection | $50 \times 2048 \times 1024$ | 180 | 17.11 | 16.73 | 1.02× |
| Expert K/V projections | $50 \times 256 \times 1024$ | 360 | 17.66 | 15.96 | 1.11× |
| Expert O projection | $50 \times 1024 \times 2048$ | 180 | 19.37 | 18.14 | 1.07× |
| FFN gate/up | $50 \times 4096 \times 1024$ | 360 | 18.32 | 16.54 | 1.11× |
| FFN down | $50 \times 1024 \times 4096$ | 180 | 19.65 | 19.04 | 1.03× |
| Action Expert input | $50 \times 1024 \times 32$ | 10 | 14.15 | 14.49 | 0.98× |
| Action Expert output | $50 \times 32 \times 1024$ | 10 | 17.37 | 16.09 | 1.08× |
| **Weighted average (1280 matmuls):** | | | | | **1.08×** |

**Analysis.** The ODE stage shows the lowest weighted speedup ($1.08\times$) due to small batch size ($M = 50$) and moderate hidden dimensions. Most layers achieve only marginal acceleration (1.02–1.11×), and the Action Expert input projection actually exhibits slight slowdown (0.98×) due to its extremely narrow output dimension ($N = 32$), consistent with the N-axis findings. Despite modest per-layer gains, the large number of matrix multiplications (1280 per observation, dominated by 10 ODE steps $\times$ 12 layers) means that even small per-operation speedups accumulate to measurable end-to-end improvement.

### F.4 Stage-Level Summary

Table 10 summarizes the performance characteristics of each model stage and provides guidance for selective quantization strategies.

Table 10: Stage-level performance summary. Weighted speedup is calculated by aggregating all matrix multiplication latencies within each stage, weighted by operation count.

| **Model Stage** | **Total Matmuls** (per observation) | **Weighted Speedup** | **Key Characteristic** |
|---|---|---|---|
| Vision encoder | 486 | 1.17× | Moderate dimensions (1152–4352) |
| KV cache prefix | 126 | **1.41×** | **Large FFN matrices (16384 outputs)** |
| ODE decoding (10 steps) | 1280 | 1.08× | Small batch ($M = 50$), narrow projections |

**Strategic Implications.**

- The **KV cache prefix stage** benefits most from W8A8 quantization and should always be quantized in production deployments.

- The **vision encoder** provides solid but not exceptional gains; quantization is recommended but not critical.
- The **ODE stage** shows modest improvement despite high operation count. Given the large number of iterations (typically 10–20 ODE steps), even the $1.08\times$ speedup translates to meaningful wall-clock savings during autoregressive decoding.

For memory-constrained deployments, a hybrid strategy could quantize only the KV cache prefix and vision encoder while leaving the ODE stage in FP16, trading a small performance regression for reduced implementation complexity.