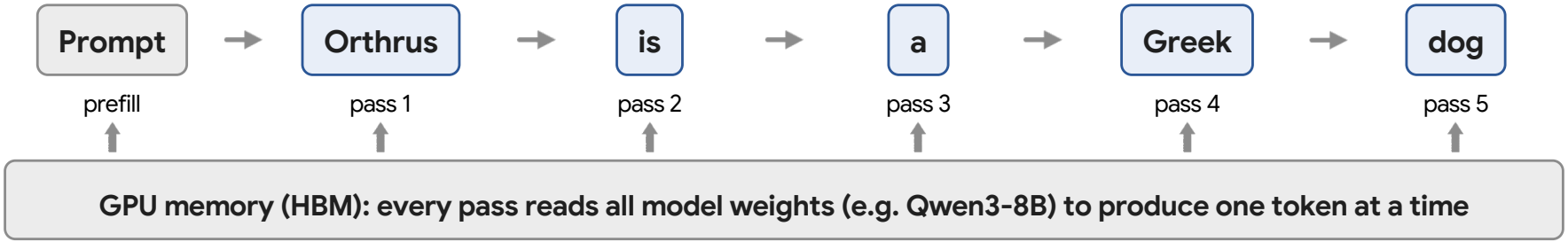
Orthrus

Memory-Efficient Parallel Token Generation via Dual-View Diffusion

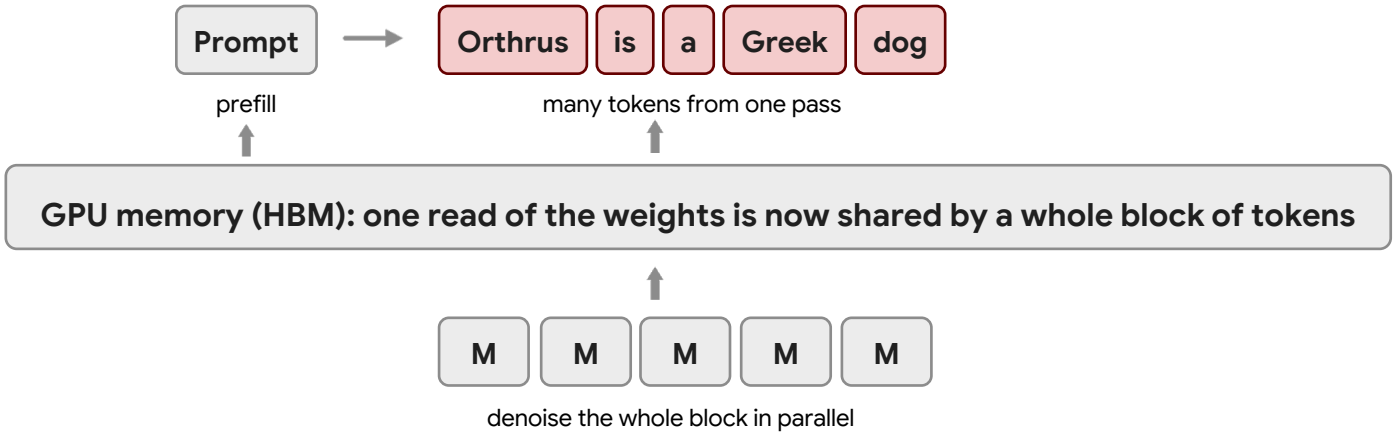
Chien Van Nguyen, Chaitra Hegde, Cuong Pham, Ryan Rossi, Franck Dernoncourt, and Thien Huu Nguyen

Motivation



LLM Decoding is heavily memory bandwidth bound. The GPU mostly waits.

Diffusion LLMs

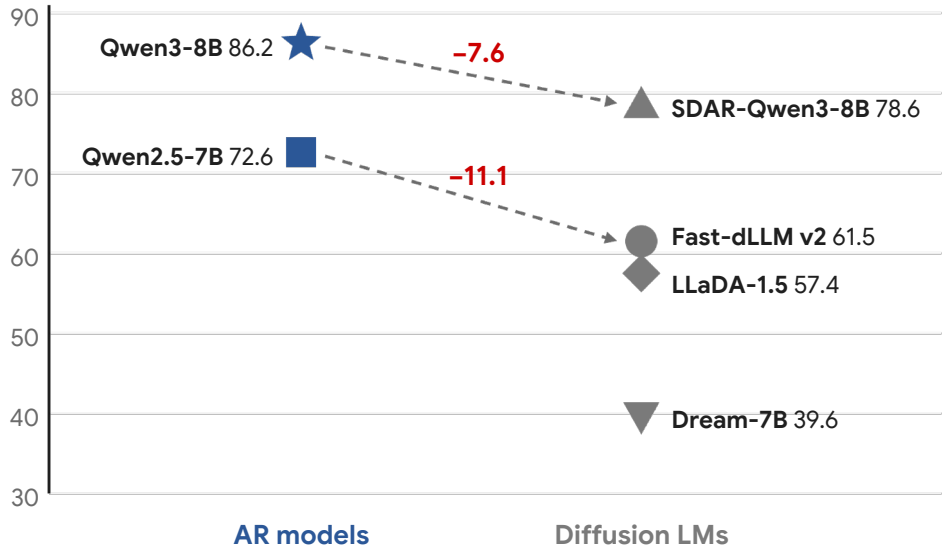


Diffusion LLMs: Decode many tokens in parallel per pass

The Quality–Speed Trade-Off

Performance on reasoning benchmarks:

Diffusion LLMs **still do not consistently match** autoregressive (AR) models.



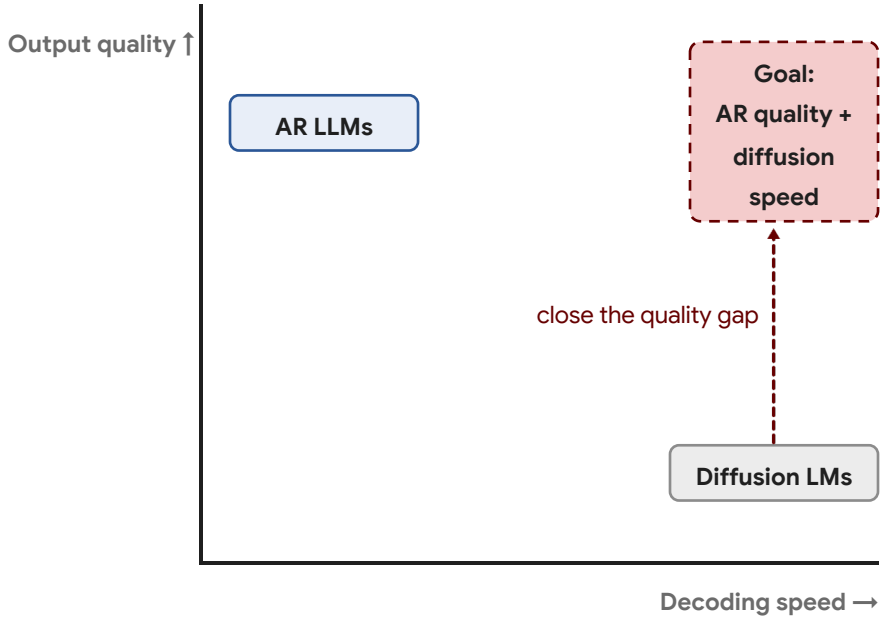
MATH-500 accuracy (%), 8B models. After diffusion adaptation.

Research Question

- **Core Assumption:** Autoregressive modeling is still the right way to go via exact AR factorization.

$$p_{\mathrm{AR}}(\mathbf{x}) = \prod_{i=1}^{N} p_\theta(x_i \mid \mathbf{x}_{<i})$$

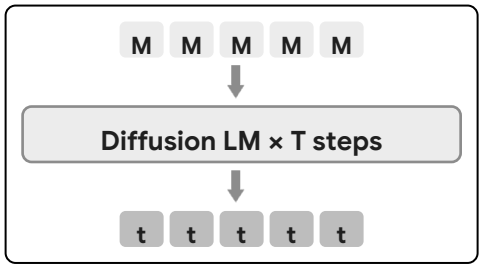
- **Can we get diffusion's parallelism with AR's exact quality?**



Current Diffusion LLMs Architectures

Pure Diffusion LM

+ Very fast parallel generation
– Quality gap, huge training cost

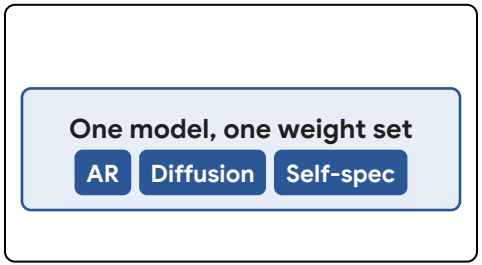


Mercury

DiffusionGemma

Multi-mode Model

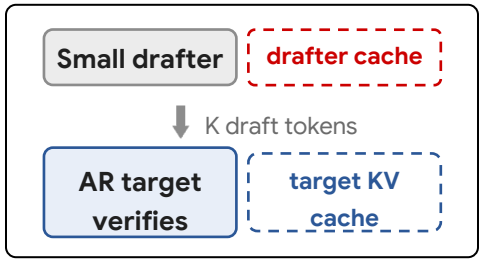
+ Better Performance
– Retrains the whole backbone



Nemotron-Diffusion, TiDAR

Speculative Decoding

+ Keeps your model; lossless
– Distributional drift → even slower



DFlash

DSpark

Limitations

Pure Diffusion LM

Big drop on reasoning:
DiffusionGemma vs. its own Gemma 4 26B-A4B base on AIME 2026: 84.2 → 69.1 (**–15.1**)

Mercury

DiffusionGemma

Multi-mode Model

Retrains the whole backbone:
- TiDAR: 150B tokens
- **Nemotron-Diffusions:** 1.4T tokens
Performance drop by 20%
Cannot guarantee lossless generation

Nemotron-Diffusion, TiDAR

Speculative Decoding

Distribution mismatch:
Drafter re-encodes context in its own state; proposals drift.

Acceptance length degrades:
DFlash, Qwen3-8B: 4.05 → 1.56 tokens accepted, 2K → 32K context

DFlash

DSpark

Our Goal: Fast, Cheap, and Memory-Efficient LLM Inference

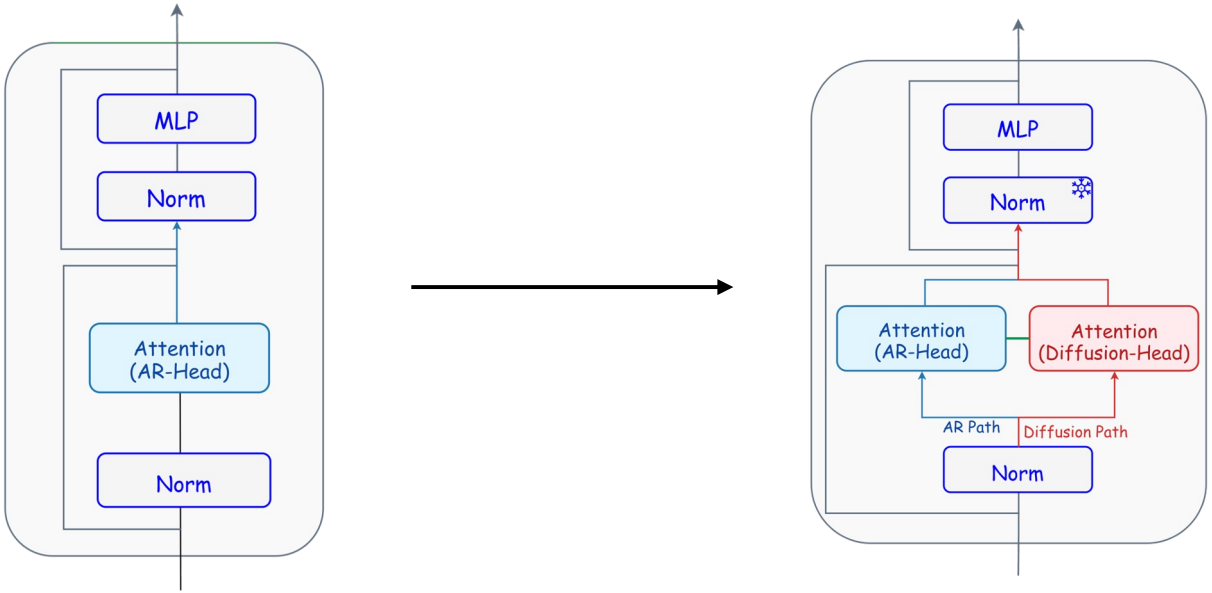
1. Faster Inference Without Quality Loss
   Accelerate decoding without sacrificing model quality.

2. Training Efficiency
   A small training budget, without retraining the entire backbone.

3. Memory Efficiency
   Minimize additional GPU memory overhead.

4. Robustness
   Deliver consistent performance across diverse tasks and context lengths.

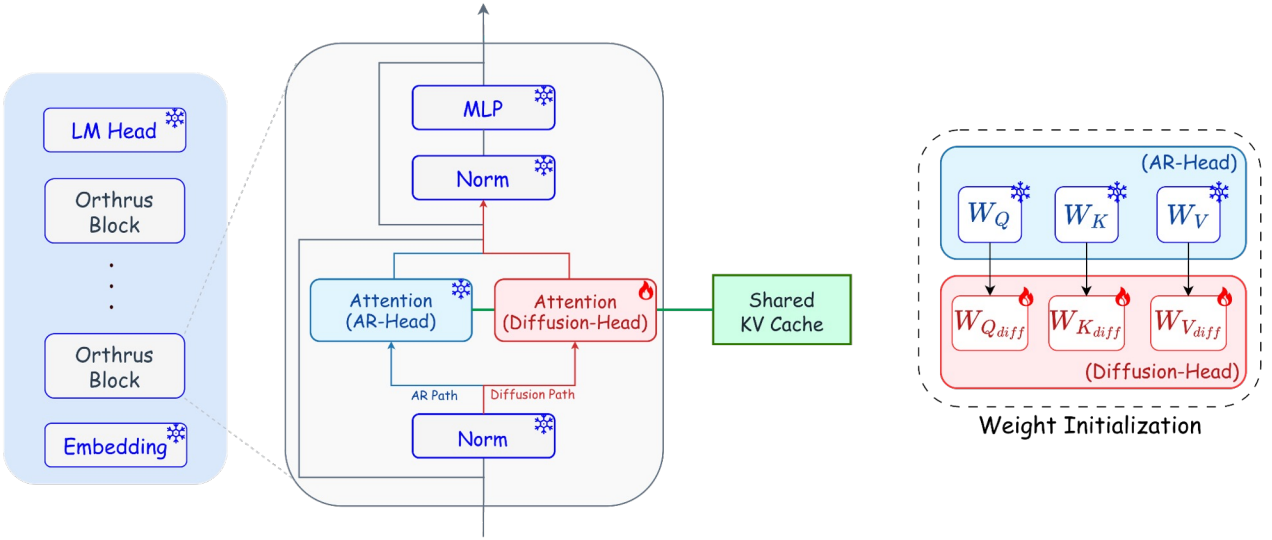
Our Approach: A Dual-View Architecture



- Augment the AR backbone with a parallel Diffusion View
- AR View for context prefilling; Diffusion View for parallel token generation

Best of both worlds: AR for lossless quality, Diffusion for parallel speed

Architecture: A second attention path inside every layer



Trainable: Diffusion Attention Modules

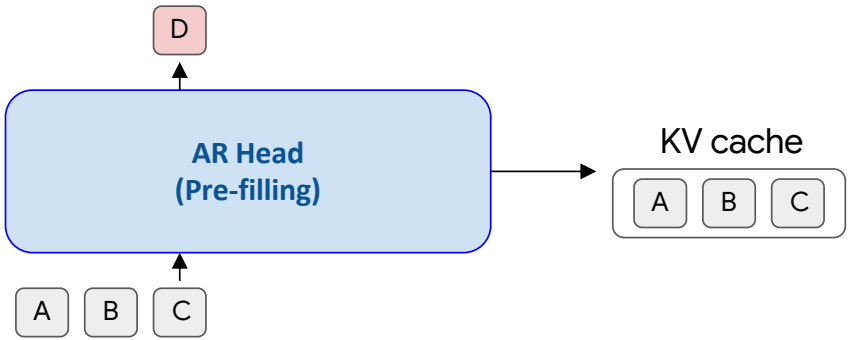
Sequence-mixer Modules for Diffusion Generator, ≈16% of parameters.

Shared Memory

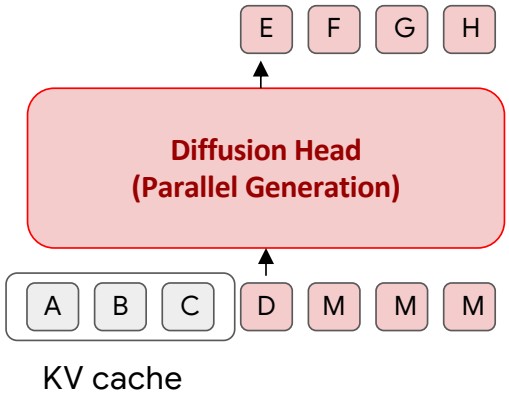
One KV cache, written only by the AR path.

Architecture: One Backbone, Two Seperate Paths

AR Path: Context-Prefilling



Diffusion-Path: Parallel Text Generation



$$\mathbf{O}_{\text{diff}} = \text{Softmax}\left(\frac{\mathbf{Q}_{\text{diff}}\,[\mathbf{K}_{\text{AR}} \,\|\, \mathbf{K}_{\text{diff}}]^{\top}}{\sqrt{d_{\text{head}}}}\right)[\mathbf{V}_{\text{AR}} \,\|\, \mathbf{V}_{\text{diff}}],$$

Design Question:

Why train only attention?

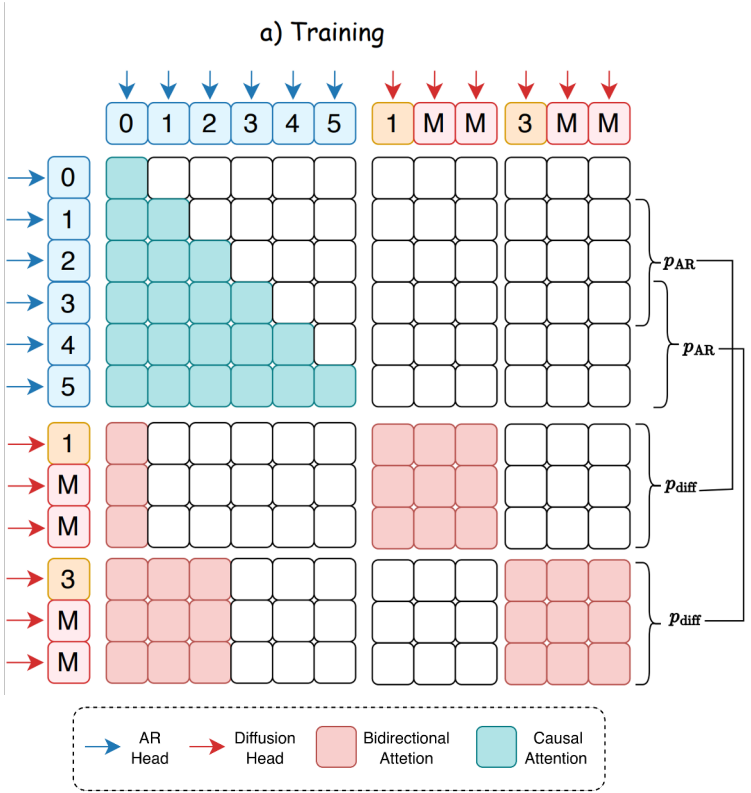
- The two views differ in exactly one thing: how information are mixed across positions. AR mixes causally; the generator mixes bidirectionally within a block.
- FFNs are just position-wise functions
- So we train the sequence mixer and freeze the rest.

QWEN3-1.7B

Trainable	Params	Accept len.	TPF
Attention only	16%	7.94	3.9
Attention + FFN	77%	7.65	3.8
FFN only	61%	3.53	1.7

5× fewer trainable parameters

Training: Distill the frozen model into the generator



Customized Mask Training:

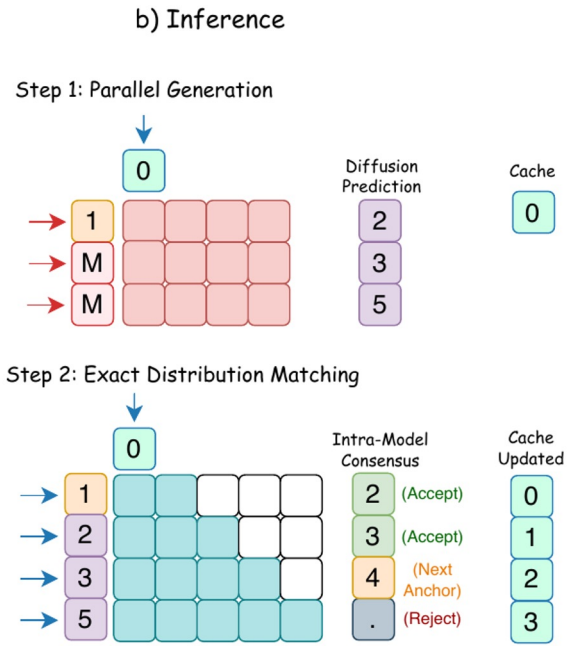
$$\mathbf{M}_{\text{diff}}[q,k] = \underbrace{\mathbf{1}[k < L] \cdot \mathbf{1}[k \le a_b - 1]}_{\text{causal AR context}} \;\mid\; \underbrace{\mathbf{1}[k \ge L] \cdot \mathbf{1}[\lfloor q/K \rfloor = \lfloor (k-L)/K \rfloor]}_{\text{bidirectional within block}},$$

Loss Function:

$$\mathcal{L}_{\text{Orthrus}} = \mathbb{E}_{\mathbf{x},\{a_b\}} \left[\sum_{b=1}^{B} \sum_{k=1}^{K} D_{\text{KL}}(p_{\text{AR}}(\cdot \mid \mathbf{x}_{\le a_b+k-1}) \,\|\, p_{\text{diff}}(\cdot \mid \mathbf{x}_{<a_b}, \tilde{\mathbf{y}}_b)) \right]$$

Training Configurations: 0.96B tokens on 2 epochs

Inference: Exact Distribution Matching



1 Draft

Diffusion view generate 32 tokens in parallel in one forward pass

2 Verify

AR view scores all 32 drafted positions at once

3 Commit

Keep drafts up to the first disagreement, **ensures output parity with the original AR**

Efficiency Benchmarking

Table 1: **Efficiency Benchmarking.** We report Tokens Per Forward Pass (TPF) and relative speedup against the sequential AR baseline (which operates at **TPF** = 1.0 and 1.0× speedup) for both greedy decoding (temperature $T = 0$) and diverse sampling (temperature $T = 1$).

Task	Qwen3-1.7B				Qwen3-4B				Qwen3-8B			
	$T = 0$		$T = 1$		$T = 0$		$T = 1$		$T = 0$		$T = 1$	
	TPF	**Speedup**	**TPF**	**Speedup**	**TPF**	**Speedup**	**TPF**	**Speedup**	**TPF**	**Speedup**	**TPF**	**Speedup**
GSM8K	4.20	4.37×	3.88	4.04×	4.91	4.48×	4.97	4.53×	5.30	5.22×	4.69	4.62×
MATH-500	4.71	4.74×	4.47	4.50×	6.02	5.76×	5.62	5.38×	6.35	5.95×	5.80	5.43×
AIME24	4.33	5.65×	3.84	5.01×	5.00	6.06×	4.70	5.70×	5.63	6.81×	4.7	5.68×
AIME25	3.89	4.80×	3.55	4.38×	5.17	5.86×	4.37	4.95×	5.25	6.03×	4.40	5.05×
HumanEval	2.75	3.07×	2.60	2.90×	3.57	3.67×	3.37	3.46×	3.94	3.68×	3.37	3.15×
MBPP	2.76	3.07×	2.88	3.20×	3.74	4.43×	3.50	4.15×	3.95	3.90×	3.75	3.70×
Pseudo2code	4.60	4.90×	4.37	4.65×	7.40	7.38×	6.95	6.93×	7.51	**7.83**×	7.29	7.60×
LiveCodeBench-v5	3.86	5.87×	3.58	5.44×	4.50	6.28×	4.28	5.97×	5.17	6.68×	4.34	5.61×
Average	**3.89**	**4.25**×	**3.65**	**4.27**×	**5.04**	**5.20**×	**4.72**	**5.13**×	**5.39**	**5.36**×	**4.43**	**5.02**×

Table 1 details these efficiency gains across our evaluation suite. Orthrus delivers substantial inference acceleration on all reasoning and algorithmic tasks, achieving an average TPF of 5.39 at the 8B parameter scale. Crucially, unlike existing DLMs that inherently trade generation quality for inference speed, Orthrus *mathematically guarantees exact distributional parity with the AR baseline*, ensuring strictly lossless acceleration.

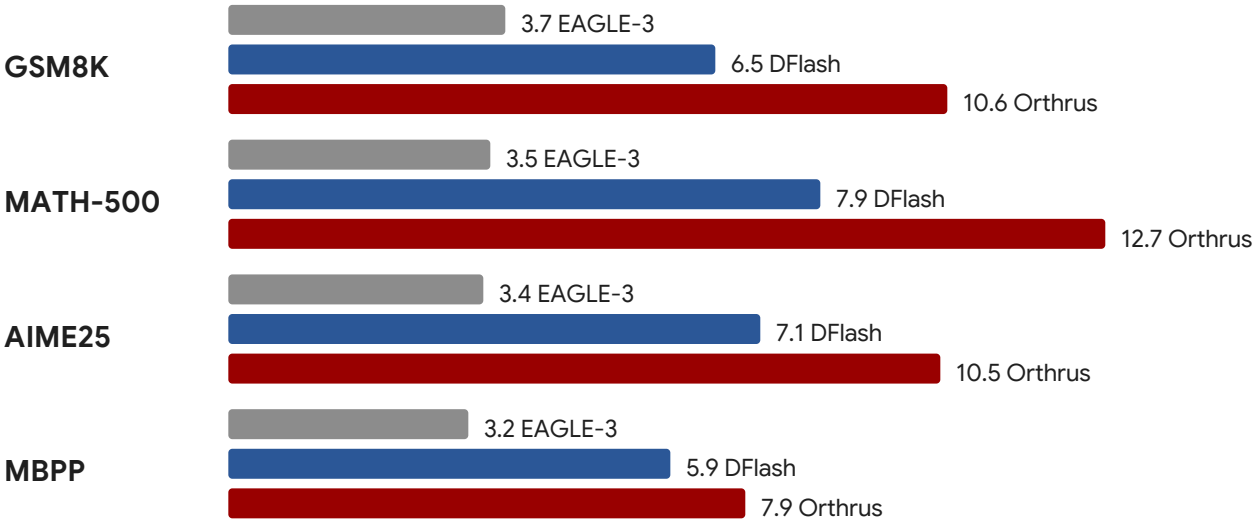
Comparison: Diffusion LLMs

Table 2: Performance comparison with SOTA Diffusion Models.

Model	Params	GSM8K	MATH-500	AIME-24	AIME-25	HumanEval	MBPP
Dream-7B	7B	79.3	39.6	-	-	57.9	58.8
LLaDA-1.5	7B	82.4	57.4	-	-	52.4	42.8
SDAR-Qwen3-8B	8B	91.7	78.6	10.0	10.0	78.7	72.0
SDAR-Qwen3-30B-A3B	30B	91.4	77.8	16.7	10.8	71.6	87.2
Fast-dLLM-v2	7B	83.7	61.5	-	-	43.3	28.2
Mercury Coder Small	-	-	-	-	-	90.0	76.6
Gemini Diffusion	-	-	-	-	20.0	89.6	76.0
Orthrus-Qwen3-8B	**8.7B**	**96.0**	**86.2**	**28.3**	**23.3**	**95.1**	**93.4**

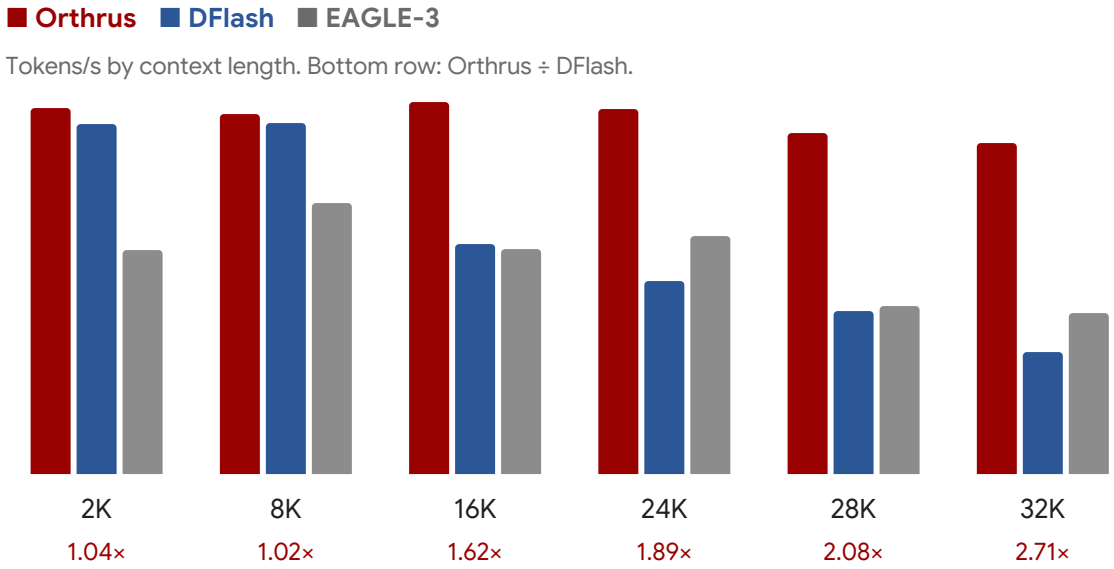
Orthrus inherits Qwen3-8B's accuracy exactly. Adaptation cost: **< 1B tokens**, vs 50B for SDAR (same Qwen3 base) and ~580B for Dream.

Comparison: Traditional Speculative Decoding



AVERAGE ACCEPTANCE LENGTH (TOKENS KEPT PER CYCLE)

Comparison: Throughput



DFlash throughput at 32K : DFlash: 102 → 35 tok/s, as its acceptance falls 4.05 → 1.56.

Continuous Batching:

Concurrency	Orthrus tok/s	AR tok/s	Speedup
1	210.6	41.2	5.11×
4	674.3	137.6	4.90×
8	763.1	206.6	3.69×
16	1,001.2	345.4	2.90×

The advantage shrinks with batch size, but persists

Generalization Beyond Transformer Architecture

Backbone	Sequence mixer	Trainable	Wall-clock speedup
Qwen3-8B	GQA attention	16%	5.36×
Llama-3.2-3B	GQA attention	13%	4.2×
Qwen3.5-4B	GQA + Gated DeltaNet	20%	5.3×

Same principle

Add a second view of the sequence mixer only; freeze everything position-wise.

Reuse the native state

Attention layers share the KV cache. DeltaNet layers share the recurrent summary and short-convolution states.

Memory Overhead

KV BYTES PER TOKEN · QWEN3-8B

36 layers × 8 KV heads × 128 dims × 2 (K,V) × 2 B = **144 KiB**

Separate drafter (DFlash/DSpark)

Keeps its own cache of the whole history: overhead grows as O(L) with context length L, and must be read every draft step.

Orthrus

Only the 32-position draft block: 32 × 144 KiB = **4.5 MiB, constant** at any L. Peak GPU memory overhead <1% (~100 MiB).

Contributions

1. Faster Inference Without Quality Loss

Exact AR distribution with up to 7.8x speedup

2. Training Efficiency

Less than 1B adaptation tokens, training within < 24 hours on 1xH200 node

3. Memory Efficiency

4. Robustness

Speedups across tasks, contexts, and architectures

Thanks for listening

Paper: arXiv:2605.12825

Code & Demo: github.com/chiennv2000/orthrus