

SUSHILA: Scalable Upstream Synthesis for Hybrid Inference in Large-model Acceleration

Up to 4× faster than Ollama on an NVIDIA A100 Open language models, benchmarked and accelerated. Get them through Sushila Station—available for Windows, MACOS and Linux

Milind K. Patil

Syncaissa Systems Inc, USA.

To Cite this Article: Milind K. Patil, “SUSHILA: Scalable Upstream Synthesis for Hybrid Inference in Large-model Acceleration”, International Journal of Scientific Research in Engineering & Technology, Volume 06, Issue 05, September-October 2026, PP:97-109.



Copyright: ©2026 This is an open access journal, and articles are distributed under the terms of the [Creative Commons Attribution License](https://creativecommons.org/licenses/by/4.0/); Which Permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

Abstract: This paper starts from existing open-source weights, stacks new inference research on top of the optimization techniques already in use, and produces faster inference of the same model—without changing a single weight. **Sushila.cpp** is a free, local inference engine built on top of **llama.cpp**, the open-source engine behind Ollama, with SGLang on NVIDIA servers. Large language models are slow and expensive to run mainly because every generated token reads the model’s weights (for mixture-of-experts models, its active experts) from memory again. **SUSHILA** (Scalable Upstream Synthesis for Hybrid Inference in Large-model Acceleration) tests the published speed-up methods on each model once, right after its release, keeps those that make it faster at unchanged quality, and adds methods of our own. One of them, a landscape precomputed once per model, previews every next-token score cheaply, then computes only the best candidates exactly, reading 13–15% of the output layer with ≥99.75% agreement on the top token (7–70B models). Over eight GPU chat and coding models, Sushila.cpp achieves a **geometric-mean speedup of 2.35×** over Ollama (**up to 4.05×** on Llama-3.3-70B, on 160 benchmark prompts it never saw, with GSM8K accuracy unchanged within its error bars), measured on an NVIDIA A100 with Sushila.cpp’s SGLang serving path. Most of this gain comes from existing methods; our precomputed draft heads add 1–11% where a good head was already published and are most of the gain where none was: **3.64×** on DeepSeek-R1-Distill-Llama-70B and **3.56×** on Kimi-Dev-72B. Speed tests were done on one NVIDIA A100 80GB for chat and coding models (media mostly on one RTX 4090; engine-level tests also on CPUs), one user at a time; results may vary on other computers. Every result has its scripts and raw outputs in our public repository, and each chat and coding comparison reruns with one command.

Keywords: efficient inference, large language models, speculative decoding, draft heads, pre-computed landscapes, output layer, mixture of experts, diffusion caching, llama.cpp, Ollama, Sushila.cpp.

Key insight

Executive Summary: run today’s best open models on your own computer, faster, for free.

What you get. Sushila.cpp is a free, open-source engine that runs chat, coding, image, music and video models on the computer you already own, and **Sushila Station** is its client app for Windows, macOS and Linux, which runs model packs in Standard or Accelerated versions on your own laptop or on a remote Sushila server. Download one file from <https://sushila.ai>, pick a model, and start: Sushila installs the right engine for your GPU (NVIDIA, AMD, Intel or Apple silicon, the CPU only as a fallback), checks every downloaded file against a signed list, and keeps your prompts, pictures, songs and videos on your machine (unless you choose a remote server). There is no subscription and no per-token bill.

Why it is faster. For every model we support, we did the expensive work once: we tested the published speed-up methods on that model, kept the ones that are really faster at the same quality, and added our own precomputed files where they help. You get the result with one switch, Standard or Accelerated, per model:

- **Chat and code:** up to **4.05× faster** than vanilla Ollama on Llama-3.3-70B, **3.64×** on DeepSeek-R1-Distill-Llama-70B, **3.56×** on Kimi-Dev-72B and **3.14×** on Qwen3-32B, with GSM8K accuracy unchanged within its error bars (geometric mean 2.35× over eight models; one A100 80GB, prompts the system never saw).
- **Images:** a 768×768 picture in **under a second** on a desktop RTX 4090 or 5090 (6 steps), and 1.3–1.7× faster than full precision at production sizes, with comparable quality.
- **Video and music:** 5-second 720p videos 1.62× faster than the same engine without our plan, and a 60-second song in about 4 seconds on an RTX 4090.

What we do not promise. The tests were done on an NVIDIA A100 80GB data-center GPU for chat and coding models and on desktop and workstation GPUs (RTX 4090/5090, L40S) for media, one user at a time; results may vary on other computers. On a laptop, gains can be smaller (an 8 GB laptop GPU makes a picture in tens of seconds, not one), and on a busy multi-user server speculation should be switched off. `sushila bench <model>` measures Standard against Accelerated on your own computer, and Standard is always one click away.

Why you can trust it. Every number in this paper follows one protocol—the baseline people run today on the same machine, prompts never used for tuning, a quality check for every speed-up—and every test, raw result and script is in our repository, <https://github.com/syncaissa/sushila.cpp>, where one command reruns each chat and coding comparison.

Try it. Sushila Station (Windows, macOS, Linux) and the headless engine `sushila` are at <https://github.com/syncaissa/sushila.cpp/tree/main/downloads> and <https://sushila.ai>.

I. INTRODUCTION

At batch size one, generating a token streams essentially all weights of a dense language model (the active experts of a mixture-of-experts model) from memory, so inference is bound by memory bandwidth rather than arithmetic [Pope et al., 2023]. Every method that reads fewer bytes per useful token makes generation faster and cheaper: quantization shrinks each weight [Dettmers et al., 2022, Frantar et al., 2023, Lin et al., 2024], speculative decoding checks several drafted tokens per pass of the model [C. Chen et al., 2023, Leviathan et al., 2023], draft heads make those drafts cheap and accurate [Cai et al., 2024, Y. Li et al., 2024, 2025], mixture-of-experts models read a few experts per token [Fedus et al., 2022, Shazeer et al., 2017], and sparsity methods skip weights whose inputs are small [J. Liu et al., 2025, Z. Liu et al., 2023].

None of these methods is best for every model. A published draft head can speed one model up 2.4× and slow its distilled sibling down; a serving engine that is 1.6× faster than Ollama on one model is slower on another (0.95× on Gemma 3); a caching plan that makes a 50-step video model 1.62× faster gives an 8-step image model only 1.10×. SUSHILA therefore makes a different contribution: **test the applicable methods on each model once, keep what helps, and add precomputed methods of our own**, all prepared right after the model’s release and bound to its file, so that every user gets the fastest measured stack without retraining anything (Figure 1).

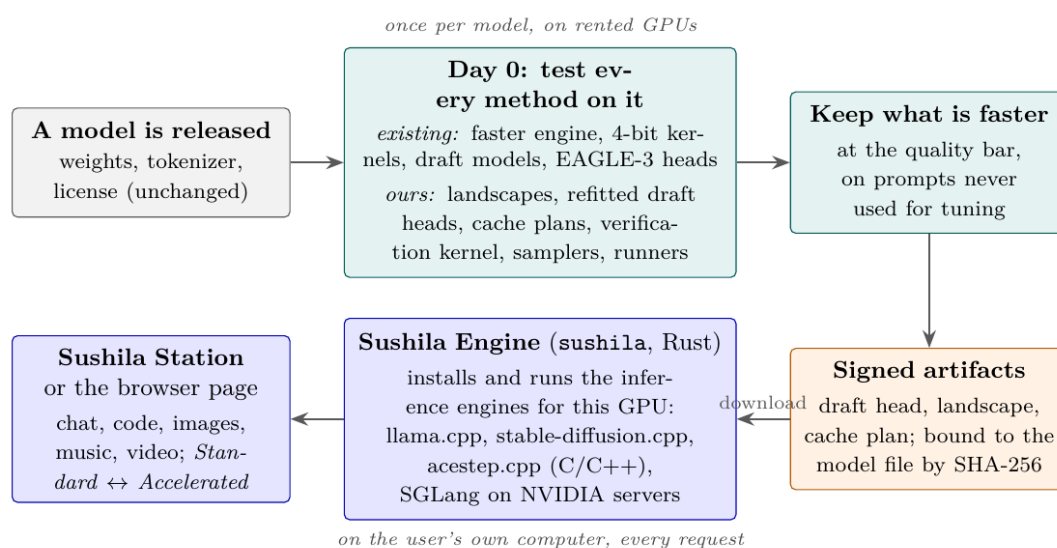


Figure 1: How SUSHILA works. Once per model (day 0), on rented GPUs, every applicable method—existing and ours—is tested and the fastest stack that keeps quality is kept as signed, checksum-bound artifacts. On the user’s computer the Sushila Engine runs that stack, with a Standard/Accelerated switch per model.

What we found, in brief. On Llama-3.3-70B the 4.05× over vanilla Ollama is the product of three measured steps: a faster engine with 4-bit kernels (1.57×) and a published EAGLE-3 draft head (2.38×), both existing work, and our precomputed draft head, refitted to the model’s own answers (1.08×). Where no good head exists our head is most of the gain: 2.30× over the engine alone on DeepSeek-R1-Distill-Llama-70B, whose base model’s head slows it down. Our other methods give smaller, measured gains. A *pre-computed landscape* finds the next token while reading 13–15% of the output layer ($\geq 99.75\%$ agreement on prose, code, chat and eight languages) and makes a small model’s CPU decoder 1.07–1.26× faster; reading 6 of 8 experts makes a mixture-of-experts model 1.11–1.14× faster at a 0.1% change in perplexity; and our verification work makes draft-head decoding in llama.cpp 1.30× faster on a GPU. For media, per-model cache plans make image and video generation 1.10× and 1.62× faster than the same engine without them (video then matches, not beats, the official pipeline), and an equivalent runner makes YuE songs 4.77× faster on the same song (research code, not yet in the product). With many simultaneous users speculation stops paying, and we show where to turn it off.

Contributions. (1) A per-model, test-and-stack pipeline, with results on 13 models and every step attributed to existing work or ours (Sections 4–5); (2) precomputed draft heads, refitted once to each model’s own answers (Section 5); (3) pre-computed

landscapes of the output layer and their CPU kernel in llama.cpp (Section 6); (4) engine work that makes speculation pay in llama.cpp, and measured MoE savings (Section 6); (5) cache plans, an exact sampler and an equivalent runner for images, video and music (Section 7); (6) one test protocol and a public replication repository for every number (Section 3).

Item	Measured effect
Precomputed draft heads (refitted on day 0)	+8% on Llama-3.3-70B, +1% on Qwen3, +9–11% on Gemma 3 and Llama-3.1-8B; 2.72× over the base model’s head on R1-Distill-Llama-70B; +41% on Kimi-Dev-72B
Output-layer landscape and its CPU kernel	–15% of the layer read with $\geq 99.75\%$ top-token agreement on every kind of text (Llama-3.1-8B/70B, Qwen2.5-7B); 1.07–1.26× faster decoding (Qwen2.5-0.5B, CPU)
Verification in llama.cpp (trees, Ampere kernel switch)	verifying 5–8 tokens 18–38% cheaper; draft-head decoding $0.95\times \rightarrow 1.30\times$
MoE expert count	of 8 experts: 1.11–1.14× at a 0.1% perplexity change (approximate output)
Cache plans, sampler, runner	images 1.10×, video 1.62×, ACE-Step 1.27×, YuE 4.77×
Faster engine and 4-bit kernels [Frantar et al., 2024, Lin et al., 2024, Zheng et al., 2024]	× over Ollama on Llama-3.3-70B (0.95× on Gemma 3: not kept alone)
EAGLE-3 draft heads [Y. Li et al., 2025]	× on Llama-3.3-70B; 0.85× (a slowdown) on R1-Distill
Small draft models [Leviathan et al., 2023]	× on Llama-3.1-70B in llama.cpp (GPU), 2.29× (CPU)
SVDQuant 4-bit image kernels [M. Li et al., 2025]	–1.72× over full precision for Z-Image-Turbo

Table 1: What is ours and what is not, with the measured effect of each.

II. HOW SUSHILA WORKS

Day 0: the hunt. When a model is released, a pipeline fingerprints it (family, tokenizer, dense or MoE, sizes), then tries, in order, the methods and landscapes that won on similar models, then the rest of the portfolio, and finally searches new settings and combinations (Algorithm 1). Each candidate is tuned on validation prompts and must meet a quality bar in every domain on a disjoint test set; the fastest one that passes is recorded in the model’s manifest. Methods attack different costs—a draft head makes passes fewer, a landscape makes each pass read fewer bytes, a better kernel makes each pass faster—so they can stack; a stack’s speedup is always measured as a whole, never assumed to be the product of its parts.

Algorithm 1 Day-0 hunt for a new model

Require: Model file M ; portfolio $\mathcal{P}$ of methods (existing and ours) with the models that chose them; quality bar q per domain

Ensure: The fastest stack L^* that meets q , recorded in M ’s manifest

- 1: Fingerprint M ; order $\mathcal{P}$: winners on the same family, then the same architecture, then the rest, then stacks
 - 2: $L^* \leftarrow$ plain decoding; $s^* \leftarrow 1$
 - 3: **for** each candidate L in order **do**
 - 4: Build L ; tune it on validation prompts
 - 5: **if** L meets q in every domain on a disjoint test set **then**
 - 6: $s \leftarrow$ speedup of L measured on the target hardware
 - 7: **if** $s > s^*$ **then** $L^* \leftarrow L$; $s^* \leftarrow s$
 - 8: **end if**
 - 9: **end if**
 - 10: **end for**
 - 11: Record L^* , its settings, its per-domain quality and s^* ; sign the artifacts; add the result to $\mathcal{P}$
-

The artifacts. What the hunt keeps is small and ships next to the unchanged model file: a draft head (a one-layer network that drafts tokens from the model’s hidden states), a landscape (a rotation and a 4-bit preview of a layer), or a cache plan (which diffusion steps may reuse earlier work). Each is bound to the exact model file by SHA-256 and listed in a signed index; a file the index does not know runs in Standard mode, so an artifact is never applied to weights it was not built and validated for.

Cheap, automatic, and strict. The day-0 pipeline (scripts/day0_landscape.sh) pins the model file’s checksum, records calibration hidden states on mixed text the model generates itself, builds and selects the cheapest setting that meets the quality bar in every domain on a *validation* set, and re-checks it on a disjoint *test* set; if any domain falls short, the model ships in exact mode, so users

are never silently degraded. It is cheap: a landscape builds in 14–21 s for Qwen2.5-0.5B on 2 CPU cores and in 2 minutes for Llama-3.1-70B on 32 cores, and the whole pipeline ran unattended in 19 minutes on 2 cores, passing its gate in every domain. The full per-model work behind the results—generating about 2,000 answers with the model, refitting its draft head, choosing the checkpoint and the complete comparison against Ollama—took about 2–7 GPU-hours per chat or coding model on one A100 80GB, and 2–4 hours per image, video or music model on one RTX 4090, on rented GPUs at \$0.74–3.18 per hour: a one-time cost per model, after which every user’s requests are accelerated.

On the user’s computer. One program, the Sushila Engine (sushila, written in Rust), installs the inference engines built for the computer’s GPU—llama.cpp for text, with our kernels; stable-diffusion.cpp for images and video; acestep.cpp for music (all C/C++); and, on NVIDIA GPUs, SGLang and a 4-bit image runtime—downloads the model packs and their artifacts, checks every file, and serves an OpenAI-compatible API and a page in the browser. The desktop app, Sushila Station (Rust with Tauri), shows the same screens. Every model has a two-position switch: *Standard* runs the plain model as Ollama or the reference engine would; *Accelerated* runs the stack the hunt chose. Section 8 describes the product.

III.HOW WE TEST

Every speedup in this paper is measured the same way, so that a reader can check it, compare it with the others and rerun it:

- 1. **The baseline is what people run today, on the same machine:** vanilla Ollama 0.35.1 on one A100 80GB for chat and coding models; stock llama.cpp on the byte-identical file for engine-level results; the reference engines on one RTX 4090 for images, video and music (production-size images on an L40S, Wan 2.2 A14B on an A100). Both run on the same GPU in the same session.
- 2. **Nothing reported was used for tuning:** artifacts are fitted on the model’s own answers and calibration text, chosen on validation prompts that are never reported, and reported on held-out prompts—26 main prompts and 160 standard-benchmark prompts (MT-Bench 80, HumanEval 40, GSM8K 40) that no artifact saw.
- 3. **End to end, wall clock:** tokens per second including prompt processing, greedy, 256 new tokens, one request at a time unless stated; 95% paired bootstrap intervals. Media: seconds per image, clip or song.
- 4. **A quality check for every speedup,** the strongest the method allows: bit-identical logits for kernels; the same output distribution for speculative decoding and samplers, with GSM8K accuracy (Wilson intervals) in both engines; top-token agreement per kind of text for landscapes; perplexity for expert counts; SSIM to the uncached output for cache plans.
- 5. **One method at a time:** methods are added one by one on the same prompts and GPU, so each step’s factor is measured on its own and the factors multiply to the measured total.
- 6. **Pinned and rerunnable:** versions, commits, model revisions and file checksums are pinned, and one command per model reruns the comparison on a rented GPU and prints our numbers beside the new ones.

Key Insight

All tests are on GitHub. Every program, prompt set, raw result and table script of this paper is in the replication repository, <https://github.com/syncaissa/sushila.cpp> (public): docs/REPRODUCE.md gives one command per model (for example bash scripts/reproduce/retest.sh llama3.3-70b), docs/REPLICATE_ALL.md the command or research script, hardware, expected numbers and pass rule for every table and figure (listed by table label, as in the paper’s extended edition), and results/ the raw outputs of every run (large binary dumps such as hidden-state arrays and model checkpoints excepted). A result counts as reproduced when its speedup is within 10% of ours.

Where these numbers hold. Every measurement holds on the machines and versions named with it. On other hardware (a laptop GPU, another GPU generation, a CPU, Apple silicon), with other versions or other settings (batch size, context length, image size), a gain may be smaller, larger or absent; sushila bench <pack> measures it on the user’s own computer.

IV.RESULTS AT A GLANCE

Table 2 lists every model with its end-to-end speedup over the usual way to run it, and the part that comes from our own precomputed work; Figure 2 separates, for each GPU chat and coding model, what the existing methods alone achieve from what Sushila.cpp achieves with our draft head. The “up to 4×” of the title is **Llama-3.3-70B**: 4.05× faster than vanilla Ollama on 160 unseen prompts (95% interval 3.98–4.11×, GSM8K accuracy 96% against 97%), from results/llama3.3-70b_20261007/summary.md in the repository, rerun with retest.sh llama3.3-70b. Over the eight GPU chat and coding models, Sushila.cpp is **2.35×** faster than vanilla Ollama as a geometric mean (arithmetic mean 2.58×). We do not average across kinds of model: the other rows use other baselines and hardware. Against a strong baseline—the same serving engine with the best published draft head, or the engine alone where that head slows the model down—Sushila.cpp is **1.01–2.30×** faster (geometric mean 1.20×) on the same eight models: 1–2% where a strong head is published (Qwen3), 8–11% over weaker heads, and 1.41–2.30× where no good head exists (Figure 2).

Model	Kind	Speedup	Ours (precomputed per model)
Llama-3.3-70B	chat	4.05×	draft head refitted to its answers: +8%
DeepSeek-R1-Distill-Llama-70B	reasoning	3.64×	draft head (none published): 2.30× over the engine
Kimi-Dev-72B	coding	3.56×	draft head from the closest published one: +41%

Model	Kind	Speedup	Ours (precomputed per model)
Qwen3-32B	chat	3.14×	draft head: +1%
Qwen3-Coder-30B-A3B	coding (MoE)	1.80×	draft head: +2%
Qwen3-30B-A3B	chat (MoE)	1.65×	draft head: +1%
Llama-3.1-8B ^a	chat	1.55×	draft head: +11%
Gemma 3 27B	chat	1.22×	draft head over a community one: +9%
Qwen2.5-0.5B	chat (CPU)	1.26×	output-layer landscape ($\geq 99.4\%$ top-1)
Z-Image-Turbo	images	1.10×	cache plan (SSIM 0.98)
Wan 2.2 TI2V-5B	video	1.62×	cache plan (frame SSIM 0.93)
ACE-Step 1.5	music	1.27×	faster exact sampler (same distribution)
YuE v1 ^b	music	4.77×	equivalent runner, same song as the official run
Geometric mean, 8 GPU chat and coding models		2.35×	arithmetic mean 2.58×; over the strong baseline (engine + best published head) 1.01–2.30×, geometric mean 1.20×

Table 2: Every model in the per-model pipeline: speedup of Sushila.cpp over the usual local baseline (text: vanilla Ollama 0.35.1 on one A100 80GB, 160 unseen prompts; Qwen2.5-0.5B: stock llama.cpp on a CPU; media: the reference engine on one RTX 4090) and the part from our own precomputed work. Text outputs are the greedy output of the file Sushila.cpp serves, up to numerical near-ties, with GSM8K accuracy unchanged within its interval. Image and video speedups are over the same engine without the plan. ^a16-bit in SGLang against Ollama’s 4-bit file. ^bResearch runner in PyTorch, not yet in Sushila.cpp.

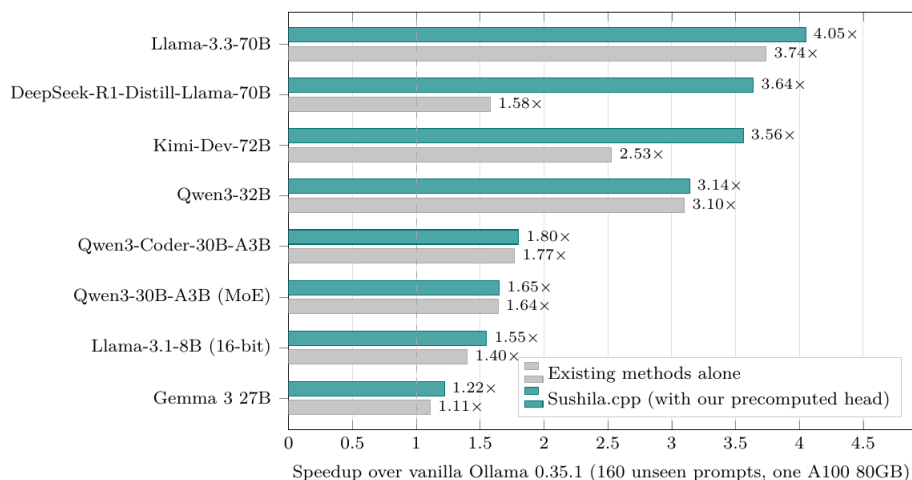


Figure 2: Existing methods alone (a faster engine with 4-bit kernels and the published draft head, or the engine alone where that head slows the model down) against Sushila.cpp with our precomputed draft head; speedup over vanilla Ollama, 160 unseen prompts, one A100 80GB.

Where a strong head was published (Qwen3) we add about 1%; where none fits (R1-Distill, Kimi-Dev) our head is most of the gain.

V.CHAT AND CODING MODELS

Where the cumulative gain comes from. Table 3 adds one method at a time on the same A100, prompts and settings. On Llama-3.3-70B a faster engine and AWQ 4-bit format give 1.57× over Ollama, a published EAGLE-3 head 2.38× on top, and our precomputed head 1.08× more: **4.05×** in total (95% interval 3.98–4.11×). DeepSeek-R1-Distill-Llama-70B shows the other end: no head has been published for it, and the head of its base model has the right shapes but predicts the distilled model’s reasoning so poorly that it slows decoding (0.85×); refitted once to 2,000 of R1’s own answers, the same head becomes 2.72× faster, for **3.64×** over Ollama.

Step	Whose	Llama-3.3-70B	Qwen3-32B	Qwen3-30B-A3B (MoE)	R1-Distill-Llama-70B	Gemma 3 27B
Vanilla Ollama 0.35.1	existing	23.6	43.2	163.8	23.5	46.5
Faster engine, 4-bit format	existing	37.2 (1.57×	68.7 (1.59×	198.7 (1.21×	37.2 (1.58×	44.3 (0.95×
Published EAGLE-3 head	existing	88.3 (2.38×	134.0 (1.95×	268.7 (1.35×	31.5 (0.85×	51.8 (1.17×

Step	Whose	Llama-3.3-70B	Qwen3-32B	Qwen3-30B-A3B (MoE)	R1-Distill-Llama-70B	Gemma 3 27B
Our precomputed head	ours	95.7 (1.08×)	135.7 (1.01×)	270.6 (1.01×)	85.6 (2.72×)	56.6 (1.09×)
Total over Ollama (95% CI)		4.05× (3.98–4.11)	3.14× (3.05–3.24)	1.65× (1.61–1.70)	3.64× (3.53–3.73)	1.22× (1.19–1.24)

Table 3: Cumulative speed, one A100 80GB, 160 prompts from MT-Bench, HumanEval and GSM8K that no draft head or selection saw; greedy, 256 tokens; tokens/s including prompt processing, with each step’s factor over the row above. Ollama reads Q4_K_M files; SGLang reads AWQ or GPTQ-Int4. R1-Distill’s “published” head is its base model’s (none exists for it). More models in the repository (docs/BENCHMARK_PIPELINE.md).

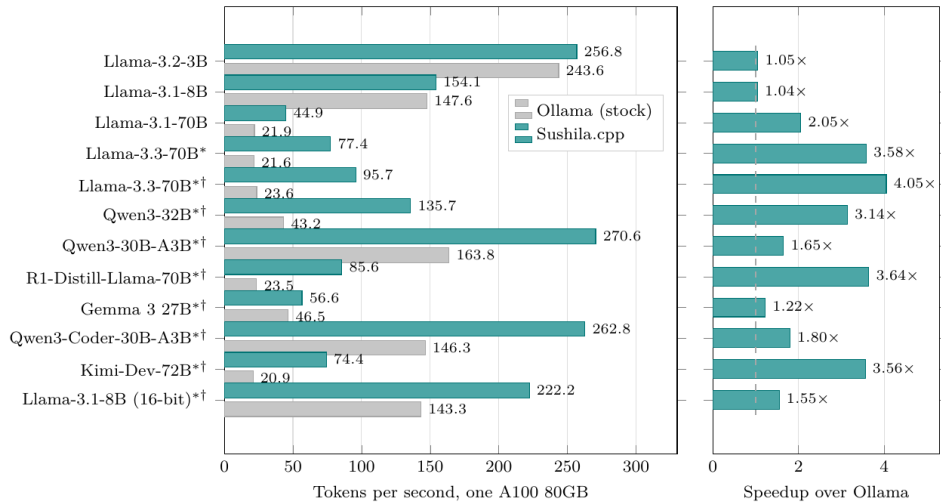


Figure 3: Vanilla Ollama 0.35.1 against Sushila.cpp on one A100 80GB. Rows 1–3: the same GGUF file and decode speed in llama.cpp (the 1B draft model for 70B, plain decoding for 3B and 8B, which a draft slows down). *Sushila.cpp’s serving path (SGLang, 4-bit AWQ or GPTQ, precomputed draft head) against Ollama’s Q4_K_M; †on the 160 unseen benchmark prompts, otherwise on the 26 main prompts.

The same answers, faster. Speculative decoding verifies every drafted token with the full model, so it returns the model’s own greedy output up to numerical near-ties, where a batched verification and a single-token step round differently. We checked both ways: GSM8K accuracy is the same in both engines within its interval for every model (Table 4), and the greedy text was word for word identical on 19–94% of the 160 prompts, the rest diverging at a near-tie. Ollama and SGLang read different 4-bit files of the same model (Q4_K_M and AWQ), so their texts differ in wording at equal accuracy. The gain depends on the task: on Llama-3.3-70B Sushila.cpp is 4.25× faster on HumanEval (code), 3.96× on GSM8K, 3.78× on MT-Bench and 3.59× when sampling at temperature 0.7 (first run, A100 PCIe); structured output is easiest to draft, and sampling keeps most of the gain.

Model	Ollama	Sushila.cpp	Speedup (95% CI)	GSM8K Ollama	GSM8K Sushila.cpp
Llama-3.3-70B	23.6	95.7	4.05× (3.98–4.11)	97%	96%
R1-Distill-Llama-70B	23.5	85.6	3.64× (3.53–3.73)	85%	85%
Kimi-Dev-72B	20.9	74.4	3.56× (3.50–3.62)	73%	72%
Qwen3-32B	43.2	135.7	3.14× (3.05–3.24)	92%	92%
Qwen3-Coder-30B-A3B	146.3	262.8	1.80× (1.70–1.89)	82%	79%
Qwen3-30B-A3B	163.8	270.6	1.65× (1.61–1.70)	91%	91%
Llama-3.1-8B (16-bit)	143.3	222.2	1.55× (1.52–1.58)	74%	76%
Gemma 3 27B	46.5	56.6	1.22× (1.19–1.24)	88%	93%

Table 4: Speed and quality on all chat and coding models: tokens/s on the 160 unseen prompts and GSM8K accuracy (100 problems, 512 tokens; R1: 40 problems, 1,536 tokens; Llama-3.3-70B: first run, A100 PCIe). Speedup of Sushila.cpp over Ollama with a 95% paired bootstrap interval.

Fitting a draft head once per model. Training a draft head on the model’s own outputs is established practice [Cai et al., 2024, SGLang Team, 2025]; what we add is doing it as part of the day-0 setup: warm-started from the closest published head, on answers rendered exactly as the model is served, with every evaluation prompt held out, and the checkpoint chosen on validation prompts that are never reported. On Llama-3.1-8B the build takes under an hour on one A100 and raises the tokens accepted per verification pass from 3.3 to 4.0 and the speedup over the engine alone from 1.86× to 2.29× (21 held-out prompts), as much on prompts unlike its training data as on held-out Dolly prompts—it learned the model, not the dataset. On Llama-3.3-70B, 1,000 answers already give 2.27–2.50× against 1.93× for the published head; more answers did not help (6,000: 2.26–2.36×), so the setup measures the head after each block of answers and keeps the best on validation prompts. Where the closest published head was trained for another model, the gain is largest: +41% on Kimi-Dev-72B, whose closest head was trained for a vision-language model of the same base.

Many users at once. Every speed above is for one request at a time—a person on their own machine. Serving engines batch many requests, which makes decoding limited by arithmetic rather than memory, and speculation, which spends extra arithmetic on drafts, then gains less or loses [X. Liu et al., 2024]. Table 5 measures total throughput with up to 64 requests in flight. Our head pays at low load on every model, and is at least as fast as the published head at every load; on the large dense models it costs throughput from 16 users on (the smaller Llama-3.1-8B still gains at 16 users, 1.29×, and loses at 64, 0.69×), so an engine should turn it off above a per-model threshold. On the mixture-of-experts Qwen3-30B-A3B our head keeps the engine’s throughput even at 64 users (1.02×), where the published head falls to 0.66×. What transfers to large-scale serving: precomputed draft heads load directly into SGLang and vLLM, to be switched off above a load threshold; reading fewer experts per token should keep its saving in every batch (counted, not timed); the output-layer landscape keeps its arithmetic saving but loses most of its byte saving, since a batch reads the union of its tokens’ candidates; and neuron skipping in dense layers does not survive batching at all.

	Llama-3.3-70B (dense)			Qwen3-30B-A3B (MoE)		
Simultaneous users	1	16	64	1	16	64
SGLang alone	37	433	862	199	1,146	2,319
Published head	88	248	245	270	1,380	1,529
Our precomputed head	96 (2.57×)	273 (0.63×)	280 (0.33×)	272 (1.37×)	1,492 (1.30×)	2,366 (1.02×)

Table 5: Throughput under load: total tokens/s across all users, one A100 80GB, SGLang, 160 benchmark prompts; in parentheses, our precomputed head relative to SGLang alone. Four more models are in the repository: Qwen3-32B, R1-Distill-Llama-70B and Kimi-Dev-72B behave like Llama-3.3-70B; Llama-3.1-8B turns from gain to loss only at 64 users.

VI. OUR OWN METHODS IN THE ENGINE

Table 6 collects what Sushila.cpp achieves in llama.cpp, the engine people run locally, on the same model file as stock: our own techniques alone give 1.07–1.30×, and with the existing small draft models that the day-0 test keeps where they help, 2.0–2.3× on Llama-3.1-70B.

Model (file)	Hardware	Stock	Sushila	Speedup	Techniques	Output
Qwen2.5-0.5B, 4-bit	CPU, 8 threads	145.5	183.3	1.26×	output-layer landscape (ours)	approx.
Qwen3-30B-A3B (MoE)	CPU, 8–32 threads			1.11–1.14×	of 8 experts (ours)	approx.
Llama-3.1-8B, 4-bit	CPU, 30 threads	19.9	24.0	1.21×	draft head + landscape inside it (ours)	exact
Llama-3.1-8B, 4-bit	A100	153	199	1.30×	draft head, tree verification, kernel switch (ours)	exact
Llama-3.1-70B, 4-bit	CPU, 30 threads	2.72	6.22	2.29×	1B draft model (existing)	exact
Llama-3.1-70B, 4-bit	A100	22.2	44.9	2.02×	1B draft model (existing)	exact

Table 6: Sushila.cpp in llama.cpp against stock llama.cpp on the same model file and hardware (greedy). Output: exact = the model’s own greedy output (speculative methods: up to near-ties); approx. = close but not token-identical.

6.1 Pre-computed Landscapes of the Output Layer

The output layer computes a score $z_v = \mathbf{E}_v \cdot \mathbf{h}$ for each of 128–262 thousand vocabulary tokens, although decoding needs only the best one or the best few; it is 9–13% of the bytes read per token in 7–30B models and 20–37% in small models with large vocabularies. A *landscape* captures, once per model, how the weights interact with the hidden states the model really produces. Let $\mathbf{C} = \mathbf{E}[\mathbf{h}\mathbf{h}^T] = \mathbf{L}\mathbf{L}^T$ be the second-moment matrix of calibration hidden states ($\mathbf{L}$ its Cholesky factor; $\mathbf{C}$ is positive definite in practice) and $\mathbf{E}\mathbf{L} = \mathbf{U}\mathbf{S}\mathbf{V}^T$; then

$$\mathbf{E}\mathbf{h} = (\mathbf{E}\mathbf{L}\mathbf{V})(\mathbf{V}^T\mathbf{L}^{-1}\mathbf{h}) = \mathbf{B}\mathbf{h}',$$

an exact rewrite in which the leading coordinates of $\mathbf{h}'$ carry most of the energy of the logits for real hidden states. At run time (Figure 4) the engine computes the first W coordinates of $\mathbf{h}'$, previews every logit from a 4-bit copy of the first W columns of $\mathbf{B}$, keeps the N best previews and computes those exactly with its own kernel. This is SVD-softmax [Shim et al., 2017] with one change that matters: the rotation is fitted to the model’s hidden states rather than to $\mathbf{E}$ alone.

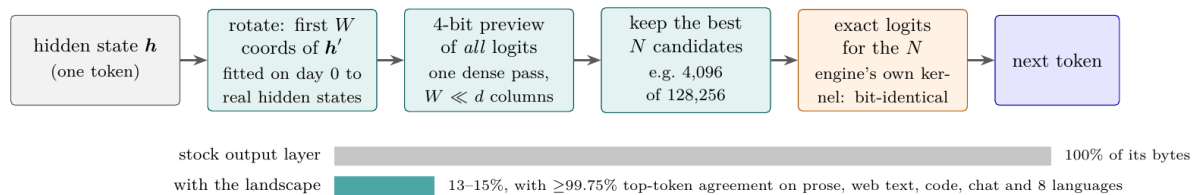


Figure 4: The output-layer landscape at run time: a cheap preview of every logit, an exact computation of a few. The candidate logits are bit-identical to the stock engine’s, so greedy decoding is unchanged whenever the true top token is among the candidates.

Results. Published SVD-softmax previews poorly on modern models, whose hidden states have outlier dimensions that the singular vectors of $\mathbf{E}$ do not see (73–97% agreement at 30% read); fitting the rotation to real hidden states fixes this (Table 7). A landscape is only as general as its calibration text: calibrated on prose alone, Llama-3.1-8B agreed on only 95% of code tokens; calibrated on mixed text—prose, C++ and Python, and chat and multilingual answers the model generated itself—it agrees on at least **99.75%**

of tokens in every domain at 13.6% of the layer’s bytes, and Llama-3.1-70B on at least 99.81% at 13.7%; for 70B, calibration matters even more: calibrated on prose alone, its landscape agreed on only 91–94% of code tokens. A pre-registered test of the calibrated bound search passed in all four model–text cells (99.7–100% at 25–29%).

Method	Llama-3.1-8B		Qwen2.5-7B	
	Top-1 (%)	Read (%)	Top-1 (%)	Read (%)
Full output layer (stock)	100	100	100	100
SVD-softmax [Shim et al., 2017], best at $\leq 30\%$ read	97.4	29.1	72.9	29.3
Preview landscape , prose calibration (prose, web)	99.9	12.7	99.95	14.7
Preview landscape , mixed calibration (every domain)	≥ 99.75	13.6	≥ 99.87	13.3

Table 7: Output layer: top-token agreement with the exact model and bytes of the layer read per token. Prose and web: 2,000 held-out tokens each (worst of the two); mixed: prose, web, code, chat and eight languages (worst domain).

Width / candidates	Calibration	Read	Prose	Web	Code	Chat	Multiling.
$W=384, N=8192$	prose only	13.6	100.0	99.95	94.84	99.70	98.44
	mixed	13.6	99.75	99.95	99.75	100.0	100.0
$W=512, N=8192$	prose only	15.9	100.0	100.0	97.07	99.78	99.11
	mixed	15.9	99.90	100.0	99.87	100.0	100.0

Table 8: Why calibration text matters: Llama-3.1-8B, top-token agreement (%) by kind of text for prose-only and mixed calibration of the same landscape. Held-out tokens: 2,000 prose, 2,000 web, 1,569 code, 1,345 chat, 1,345 multilingual.

A real kernel. We implemented the landscape as a CPU kernel inside llama.cpp [Gerganov et al., 2023], the engine behind Ollama [Ollama, 2023]. Candidate logits use the engine’s own dot product, so they are bit-identical to the stock engine’s, and greedy generations on Qwen2.5-0.5B were byte-identical to stock on our generation test prompts; on 20 further held-out prompts, the engine’s own exact recomputation of every logit gave 100% top-1 agreement on prose, chat and multilingual text and 99.4% on code, so outputs can differ where a miss occurs. Decoding is **1.07–1.26 $\times$ faster** at 1 to 32 threads on a 32-core CPU (best 1.26 $\times$ at 8 threads, 145.5 to 183.3 tokens/s). For large dense models the output layer is only about 2% of the bytes per token, so their gains must come from other layers; algorithms for attention and feed-forward layers are the next additions to the portfolio.

6.2 Making Speculation Pay in llama.cpp

Existing methods do not automatically pay in the engine people run locally. In stock llama.cpp, which drafted a single chain of tokens, the published EAGLE-3 heads were accepted on only 18–35% of drafted tokens on Llama-3.3-70B (1.09–1.17 $\times$) and gave 1.21 $\times$ on a 4-bit Llama-3.1-8B; a 1B draft model gave 2.02 $\times$ on Llama-3.1-70B on a GPU and 2.29 $\times$ on a CPU, but slowed Llama-3.2-3B and Llama-3.1-8B down (0.65–0.86 $\times$), so the day-0 test keeps plain decoding for those. Draft heads pay only if verifying B drafted tokens costs about as much as decoding one. In llama.cpp on an A100 it did not: verifying 6 tokens cost 2.1 single-token steps and 8 tokens 2.8, because on the A100’s architecture no switch point from the vector kernel to the tensor-core kernel had been tuned. With the switch measured (tensor cores from 5 tokens on) verifying 5–8 tokens costs **18–38% less**, and with tree verification—every branch of a draft tree is a sequence of the engine’s cache, so one pass verifies the whole tree—draft-head decoding becomes **1.30 $\times$ faster** than stock on Llama-3.1-8B, where it was slightly slower before (0.89–0.95 $\times$). Table 9 gives the numbers. On a CPU, a landscape built for the draft head’s own output layer cuts each draft step by about 45% and turns a head that merely breaks even into a 1.21 $\times$ faster decoder, with the model’s greedy output.

Switch	$B = 1$	$B = 4$	$B = 5$	$B = 6$	$B = 8$	7-token tree
Stock (vector kernel up to 8 tokens)	6.67	10.97	13.05	13.98	18.55	0.95 $\times$
Ours (vector kernel up to 4)	6.69	10.73	10.73	11.19	11.44	1.30$\times$

Table 9: Verification cost in llama.cpp on one A100 80GB (Llama-3.1-8B, 4-bit): milliseconds per decode step of B tokens with the stock kernel switch and ours, and the resulting draft-head decoding speed over stock (median of 4 prompts).

6.3 Mixture-of-Experts Models

In Qwen3-30B-A3B (128 experts per layer, 8 active) the active experts are 57% of the bytes read per token. Reading 6 of the 8 experts a token selects changes perplexity by 0.1% (7.540 to 7.548 on WikiText-2, 20 chunks) and decodes **1.11–1.14 $\times$ faster** on a CPU; like any change to the experts used it is an approximation, close in quality but not token-identical. Inside each expert, keeping each token’s most important half of the neurons costs 0.5% perplexity if the right neurons are known in advance, and—unlike neuron skipping in dense models—by our count of the rows a batch reads, these savings would survive batching (a batch of 16 touches about 45 of 128 experts). The neuron level assumes an oracle; the cheap predictors we tried do not reach it (Section 10).

Setting	Expert bytes read	Perplexity
Stock (8 experts per token)	100%	7.540
6 of 8 experts (shipped)	75%	7.548
8 experts, 50% of neurons in each (oracle)	50%	7.578
6 of 8 experts, 70% of neurons (oracle)	53%	7.535

Table 10: Qwen3-30B-A3B: WikiText-2 perplexity when each token reads fewer experts, or fewer neurons inside each expert (neurons ranked by their true contribution, an oracle).

VII.IMAGES, VIDEO AND MUSIC

The same once-per-model approach covers media models (Figure 5).

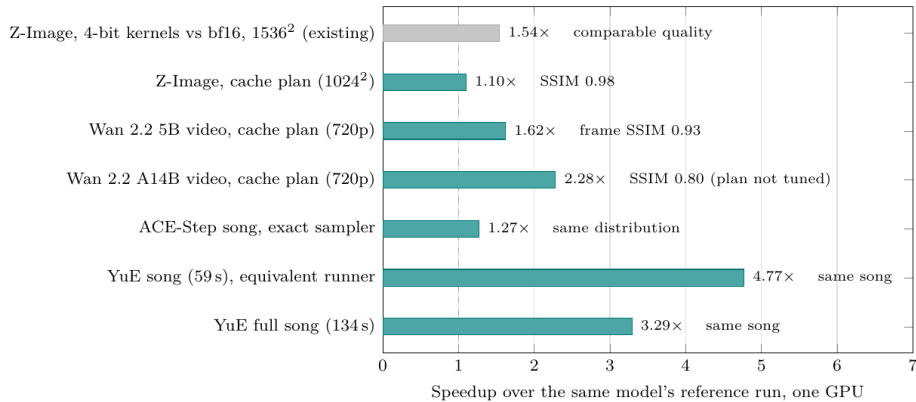


Figure 5: Media speedups, each with its quality check: SSIM to the uncached output for cache plans, the same distribution for the sampler, the same song for the YuE runner, and no-reference quality and prompt match for the 4-bit image kernels (existing work, chosen per GPU). One RTX 4090, L40S (production images) or A100 (Wan A14B).

Images in under a second. For Z-Image-Turbo [Tongyi Lab, 2025], the engine every computer gets is stable-diffusion.cpp [leejet & contributors, 2025] on a 4-bit GGUF file; on NVIDIA GPUs Sushila.cpp instead installs the existing SVDQuant 4-bit kernels [M. Li et al., 2025], chosen by compute capability (int4 for RTX 20–40, FP4 for RTX 50). A 768² image takes **0.92 s on an RTX 4090** and **0.71 s on an RTX 5090** with the Accelerated default of 6 steps, against 3.03 s for the standard engine at the model’s 8 steps; at equal steps and 1024² the times are 2.43 s and 5.30 s. At production sizes, against the official pipeline in full precision on an L40S, the 4-bit kernels are 1.72× faster at 768², 1.54× at 1536² and 1.34× at 2048², with slightly lower no-reference quality (0.3–1.9 MUSIQ points) and equal or better prompt match, and they fit a 2048² image in 22.6 GB where full precision needs 30.9 GB. These times need a GPU that holds the pipeline; on 8–16 GB laptop GPUs an image takes tens of seconds. Because a distilled model maps most seeds to similar pictures, the NVIDIA runtime adds a small seeded nudge to the prompt in the first step, which makes different seeds somewhat more different (CLIP image similarity between seeds 0.948 to 0.918) at no cost in time.

GPU, engine	768 ² , 6 st.	768 ² , 8 st.	1024 ²	Size	Speedup	Memory (GB)	MUSIQ
						std / ours	std / ours
RTX 4090, standard (4-bit GGUF)		3.03	5.30	768 ²	1.72×	22.1 / 13.8	73.8 / 71.9
RTX 4090, Accelerated (int4)	0.92		2.43	1024 ²	1.65×	23.3 / 14.9	72.6 / 72.3
RTX 5090, Accelerated (FP4)	0.71		1.85	1536 ²	1.54×	26.5 / 18.1	69.9 / 68.1
				2048 ²	1.34×	30.9 / 22.6	60.9 / 59.8

Table 11: Z-Image-Turbo. Left: seconds per image, installed as a user installs it (median of 5 prompts). Right: production sizes on one L40S against the official full-precision pipeline (nine steps, ten prompts): speedup, peak memory and no-reference quality (MUSIQ, 0–100; higher is better).

Our method: a precomputed cache plan. Diffusion models refine an image or video over many steps, and consecutive steps often change little. The day-0 pipeline tries each caching mode of the engine at several settings on calibration prompts, keeps the fastest whose outputs stay close to the uncached ones (mean SSIM), and reports it on held-out prompts (Table 12). For an 8-step image model the gain is small (1.10×); for Wan 2.2 TI2V-5B at 720p and 50 steps it is **1.62×** (889 s to 549 s for a 5-second clip), with frames that keep the same scene and motion. Honestly stated, both are over our own engine without the plan, not over the fastest pipeline: Wan’s official pipeline takes 536 s on the same prompt, so with the plan our engine matches it. On the larger Wan 2.2 A14B at 720p the reused plan gives 2.28× at a visible cost (SSIM 0.80), because it was not recalibrated for that model.

Model	Plan chosen	Uncached	With the plan	SSIM
Z-Image-Turbo, 1024 ² , 8 steps	EasyCache, threshold 0.2	5.3 s	1.10× faster	0.98
Wan 2.2 TI2V-5B, 1280×704, 5 s, 50 steps	EasyCache, threshold 0.2	889 s	1.62× faster	0.93

Table 12: Precomputed cache plans (ours), stable-diffusion.cpp on one RTX 4090; same prompt and seed with and without the plan; held-out prompts. SSIM: similarity to the uncached output (1 = identical).

Music. ACE-Step 1.5 spends most of its time writing a song plan with a 4B language model, and its engine sorted all 65,536 code probabilities on the CPU for every token. Our sampler finds the top-*p* boundary from a histogram and sorts only the boundary bin: the same distribution (largest total-variation difference 8×10^{-4}), and a 60-second song **1.27×** faster (5.16 to 4.05 s). Speculative sampling did not pay here: the codes are drawn at high entropy, which no cheap drafter matches. For long songs, YuE’s official code re-runs its second model on the whole growing sequence for every audio frame. Our runner keeps one cache across frames, batches rows and the guidance branch, and uses CUDA graphs whose logits equal eager PyTorch’s exactly (Table 13). Replaying the official run’s tokens so that both produce the same song, it is **4.77×** faster on a 59-second song and 3.29× on the full 134-

second song; with its second stage in float32 it reproduces the official float32 output code for code. The runner is research code, not yet in Sushila.cpp.

Runner	Stage 1	Stage 2	Total	Speedup	Output
Official infer.py, 59 s song	370	822	1,210	1.00×	reference
Ours , 59 s song	182	57	253	4.77×	same stage 1; bfloat16 near-ties in stage 2
Ours, stage 2 in float32	186	195	396	3.06×	every code equal to the official float32 run
Official, full 134 s song	840	1,493	2,353	1.00×	reference
Ours , full song	554	141	715	3.29×	same stage 1

Table 13: YuE v1 on the same song (stage-1 tokens replayed from the official run), one RTX 4090, seconds.

VIII.SUSHILA.CPP AND SUSHILA STATION: THE PRODUCT

Sushila Station is a client application for inference, for Windows, macOS and Unix-like systems (Linux): it runs the model packs of Sushila’s catalog—chat, code, images, music and video—in their *Standard* and *Accelerated* versions, locally on the user’s own laptop or desktop, or using models installed on a remote server running the Sushila Engine, reached through that server’s internet link. Builds are provided for Windows, macOS and Linux; remote use today means another computer running the Sushila Engine (connections to other model servers are planned).

The methods above reach users as one program and one app. The **Sushila Engine** (sushila, Rust) has the same commands on every system (Windows, macOS and Linux); it installs the engine build for the computer’s GPU automatically (CUDA, Vulkan, Metal; the CPU only as a fallback, and its self-test fails if a GPU is present but unused), installs verified model packs (signed indexes, every file checked by SHA-256), runs chat, code, image, music and video models with a background queue, and serves an OpenAI-compatible API and a browser page. **Sushila Station** is the desktop app (Rust with Tauri; the same screens as the page). Releases of both are signed (Ed25519 over the application, build, system, checksum and size) and update themselves in place after all checks pass; on a user’s Windows laptop the app upgraded itself through several builds without reinstalling. Model packs are rows of a catalog that records each pack’s files, upstream source and precomputed artifacts, so packs are added or withdrawn without a new program. The catalog does not take every model that is released: we add the models that run well on computers people can own—a laptop, a desktop or a single workstation GPU—and leave out those that need a data-center server, such as Qwen3-235B-A22B, which needs two 80 GB GPUs and where speculation slowed decoding to $0.73\times$ Ollama’s speed in our tests (SGLang alone $1.25\times$). Users can also install a model that is not in the list from a Hugging Face, GitHub or Dropbox link (data formats only); such a model has no precomputed artifacts and runs in Standard mode. Precomputed artifacts can reach users in three ways: *bundled* with the model in a pack; *bring your own weights*, where the engine computes the checksum of a model file the user downloaded from its official source and fetches the artifacts built for exactly that file; and *on demand*, where a fine-tuned or private model is sent through the same day-0 pipeline once and its artifacts are stored under its checksum. In every route a file the service does not know runs in Standard mode, and because artifacts are a few percent of the model’s size, the weights stay with their owners. Everything runs locally and costs nothing per token; prompts and outputs stay on the computer unless the user chooses a remote server or shares a file. At each start of the program or the app, and at each load of the browser page, a small license check (app, version, operating system, start or page load, and the signed-in account if there is one; the server records the license type—“Free” for the free download—and the address and country it comes from) is sent to sushila.ai for audit and license enforcement; it carries no prompts or content, and the code that sends it is public.

IX.RELATED WORK

Speculative decoding [C. Chen et al., 2023, Leviathan et al., 2023] and **draft heads** [Cai et al., 2024, Y. Li et al., 2024, 2025] preserve the output distribution and share each pass’s weight read over several tokens; we adopt them and fit the head once per model. Drafting without a separate model—lookahead decoding [Fu et al., 2024], prompt lookup [Saxena, 2023], retrieval [He et al., 2024] and self-speculation from the model’s own early layers [Elhoushi et al., 2024]—are further candidates for the same per-model test; in our measurements lookup drafting did not pay on open chat. **Quantization** [Dettmers et al., 2022, Frantar et al., 2023, M. Li et al., 2025, Lin et al., 2024] shrinks every weight but changes the model file; landscapes are built per quantized file. **Activation sparsity** [J. Liu et al., 2025, Z. Liu et al., 2023] skips weights per token, with learned predictors or magnitude thresholds, evaluated on aggregate benchmarks rather than against a per-domain error budget; our MoE results show where such savings survive batching. **SVD-softmax** [Shim et al., 2017] previews the output layer from the singular vectors of the weights alone, which previews poorly on models with outlier dimensions (73–97% top-token agreement at 30% read in our tests); fitting the rotation to real hidden states is what makes it work. **Serving engines** [Kwon et al., 2023, Zheng et al., 2024] and **local engines** [Gerganov et al., 2023, Ollama, 2023] are the platforms we build on and measure against; adaptive speculation under load [X. Liu et al., 2024] is how an engine should use our heads at high concurrency. Choosing the fastest implementation per model and hardware by measurement is also the idea behind compiler auto-tuning [T. Chen et al., 2018]; SUSHILA applies it one level up, to whole inference methods and their precomputed artifacts. SUSHILA’s distinct contribution is the combination: a per-model test of all of these, once per release, with precomputed artifacts bound to the model file and a public protocol for every number.

X. WHAT DID NOT WORK, AND LIMITATIONS

What did not work. Sampling each layer’s sum at run time, where SUSHILA started, is unbiased but loses to simply dropping small terms (Appendix B). Skipping feed-forward neurons per token in dense models does not survive batching, since tokens choose their neurons almost independently. Predicting the important neurons inside experts from the gate projection or from low-rank sketches fell well short of the oracle. In llama.cpp, small same-family draft models (0.5–0.6B) slowed most models down (0.28–

0.90×), so they stay off where the day-0 test finds them slower. An input-sparse kernel made a good self-draft for Llama-3.1-70B (82–96% of tokens accepted) but its matrix products were too slow to pay on a CPU. A draft refitted to YuE’s own songs, and n-gram lookup drafting on open chat, did not raise acceptance enough to pay.

Limitations. All speeds are for the GPUs and versions named with them; laptop GPUs, other generations and Apple silicon were not measured systematically. Text timings are single requests with prompts of 41–384 tokens and outputs of 256–1,536, so long contexts are untested, and speculation should be turned off at high load (Table 5). Ollama and Sushila.cpp read different 4-bit files on NVIDIA GPUs (Q4_K_M and AWQ), so their texts differ in wording at equal accuracy, and the AWQ files are 6% smaller, which is part of the engine’s gain. The output-layer landscape, the expert count and the cache plans are measured approximations, not exact methods, and the user can switch to Standard at any time.

XI.WHAT THIS CHANGES

When decoding is bound by memory traffic, cost and speed follow the bytes read per token, so every saving here is also a saving in money and energy. (1) *Local and private AI*: models up to about 32B parameters, images, songs and short videos run on the user’s own computer at no cost per token, with prompts and outputs that stay on it; small models with large vocabularies, typical of laptops and phones, are exactly where the output-layer landscape gains most. (2) *Hosted inference*: mixture-of-experts models are common in large deployments, and their expert savings, by our count, survive batching. (3) *Energy*: moving fewer bytes is a major lever on the energy of memory-bound inference. What a single machine cannot do well—models far larger than its memory, long high-resolution video, many simultaneous users—still belongs on servers; every speedup here widens what stays local.

XII.CONCLUSION

Fast local inference does not need a new model, only the right methods for each model, chosen once. Testing the published methods per model and stacking the ones that help makes Llama-3.3-70B 4.05× faster than vanilla Ollama and eight chat and coding models 2.35× faster (geometric mean), with GSM8K accuracy unchanged within its interval; our precomputed draft heads add little where a strong head exists and most of the gain where none does (3.64× and 3.56×); and our own engine methods give measured, smaller gains in the engine people run locally (1.07–1.30×) and for media (1.10–1.62× in the product, 4.77× in research code). Every number follows one protocol and has its scripts and raw outputs in the public repository. The next steps are clear: draft heads for every new model on its release day, a verification kernel whose cost stays near one step for large token trees in llama.cpp, landscapes for attention and feed-forward layers, and measured gains on laptop-class GPUs.

Code, Results and Downloads

Replication repository (every test, raw result and script): <https://github.com/syncaissa/sushila.cpp>
Sushila Station and sushila.cpp downloads: <https://sushila.ai>

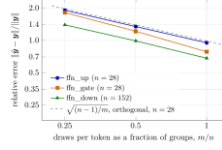
XIII.REPRODUCE EVERY RESULT

Every result in this paper is rerun from the replication repository <https://github.com/syncaissa/sushila.cpp> with the commands below (from the repository root). docs/REPLICATE_ALL.md adds the time, the expected numbers and the folder of raw results for every table and figure, by table label; docs/REPRODUCE.md pins every software version, model revision and file checksum. A result counts as reproduced when its speedup is within 10% of ours. *Known gaps*, listed in that file: the landscape, kernel and MoE results come from the original research scripts, copied unchanged with paths of our machines; large binary dumps were not copied. Three numbers that were one-off runs now have rerun scripts that reproduce them: the video plan’s per-prompt frame SSIM (scripts/video/video_quality_ssim.sh), the ACE-Step sampler’s distribution check (scripts/music/test_fast_sampler.sh), and the official YuE float32 reference (scripts/music/yue/run_yue_official_fp32.sh; rerun on 2026-10-09: every float32 stage-2 code of ours equals the official float32 run’s, as in Table 13; official float32 1,635 s against our 391 s). Media results are single runs per prompt, and the build hours and costs of Section 2 are accounting from our pod logs, not a measurement.

Result	Command	Hardware
Tables 2, 3, 4, 5; Figures 2, 3	bash scripts/reproduce/retest.sh <model> (LOAD=1 for load; FULL=1 rebuilds our head)	× A100 80GB
Draft heads, Llama-3.3-70B (Section 5)	cd scripts/bench70b; bash run_all.sh	× A100 80GB
Tables 7, 8	scripts/day0_landscape.sh <model>	CPU, 32 vCPU
Tables 9, 6, 10	the original research scripts in scripts/research/ (pod2_kernel.sh, pod_moe.sh, ...; see docs/REPLICATE_ALL.md)	CPU; 1× A100
Table 11	bash scripts/image/bench_zimage.sh; bash scripts/image/bench_zimage_production.sh	RTX 4090/5090; L40S
Table 12	python3 scripts/cache/calibrate_cache_plan.py; bash scripts/video/run_wan_720p.sh	RTX 4090
Table 13	bash scripts/music/yue/reproduce_yue.sh	RTX 4090
Seed diversity (Section 7)	scripts/image_seed_diversity/	RTX 4090
Model files and artifacts	python3 scripts/reproduce/verify_signature.py <model>	any

Table 14: How to rerun each result.

XIV. WHERE SUSHILA STARTED: MONTE CARLO SAMPLING



Tail treatment (Llama-3.1-8B)	Perplexity
stock engine	7.91
placebo: matched noise	1.29×10^6
mc: importance-sampled	1.41×10^6
zeros: dropped	4.73×10^5
topk: next-largest exact	4.67×10^5

Figure 6: Left: the error of pure importance sampling follows the $1/\sqrt{m}$ law (Qwen2.5-0.5B). Right: placebo-controlled ladder, Llama-3.1-8B, 10% of feed-forward column groups read (mean of 3 seeds for mc, placebo).

SUSHILA began as an attempt to make inference cheaper by *sampling*. Split a layer’s input columns into groups g with contributions $\mathbf{c}_g = \mathbf{W}_{:,g} \mathbf{x}_g$, so $\mathbf{y} = \sum_g \mathbf{c}_g$; score each group by $s_g = \|\mathbf{x}_g\| \|\mathbf{W}_{:,g}\|$; compute the highest-scoring groups (the head $\mathcal{H}$) exactly and estimate the rest (the tail $\mathcal{T}$) from m importance-weighted draws:

$$\hat{\mathbf{y}} = \sum_{g \in \mathcal{H}} \mathbf{c}_g + \frac{1}{m} \sum_{j=1}^m \frac{\mathbf{c}_{g_j}}{p_{g_j}}, \quad g_j \sim p, \quad p_g \propto s_g.$$

Theorem 14.1 (Unbiased, but noisy). *If $p_g > 0$ wherever $\mathbf{c}_g \neq \mathbf{0}$, then $\mathbb{E}[\hat{\mathbf{y}}] = \mathbf{W}\mathbf{x}$, and for any sampling distribution $\mathbb{E}\|\hat{\mathbf{y}} - \mathbf{y}\|^2 \geq \frac{\rho_{\mathcal{T}} - 1}{m} \|\mathbf{y}_{\mathcal{T}}\|^2$, where $\rho_{\mathcal{T}} = (\sum_{g \in \mathcal{T}} \|\mathbf{c}_g\|)^2 / \|\mathbf{y}_{\mathcal{T}}\|^2 \geq 1$ measures how much the tail’s contributions cancel. Dropping the tail has squared error exactly $\|\mathbf{y}_{\mathcal{T}}\|^2$, so sampling can win only if $m > \rho_{\mathcal{T}} - 1$.*

Proof sketch. The head is exact, and each draw $\mathbf{c}_{g_j}/p_{g_j}$ has mean $\mathbf{y}_{\mathcal{T}}$, so $\hat{\mathbf{y}}$ is unbiased. For m independent draws, $\mathbb{E}\|\hat{\mathbf{y}} - \mathbf{y}\|^2 = \frac{1}{m} (\sum_{g \in \mathcal{T}} \|\mathbf{c}_g\|^2 / p_g - \|\mathbf{y}_{\mathcal{T}}\|^2)$; by Cauchy–Schwarz the sum is at least $(\sum_{g \in \mathcal{T}} \|\mathbf{c}_g\|)^2$, with equality at $p_g \propto \|\mathbf{c}_g\|$. $\square$

In a transformer the column contributions point in nearly unrelated directions, so $\rho_{\mathcal{T}}$ is close to the effective number of tail groups (their number, when their norms are similar) and sampling must read about as much as computing the tail. The measurements agree (Figure 6): the error follows the $1/\sqrt{m}$ law, and importance sampling (mc) was no better than matched noise (placebo) and far worse than dropping the tail (zeros). That negative result is why SUSHILA moved to *precomputing*, once per model, what each token needs.

REFERENCES

- Cai, T. et al. (2024). Medusa: Simple LLM inference acceleration framework with multiple decoding heads. International Conference on Machine Learning (ICML).
- Chen, C. et al. (2023). Accelerating large language model decoding with speculative sampling. arXiv:2302.01318.
- Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., Guestrin, C., & Krishnamurthy, A. (2018). TVM: An automated end-to-end optimizing compiler for deep learning. USENIX Symposium on Operating Systems Design and Implementation (OSDI).
- Dettmers, T., Lewis, M., Belkada, Y., & Zettlemoyer, L. (2022). LLM.int8(): 8-bit matrix multiplication for transformers at scale. Advances in Neural Information Processing Systems, 35.
- Elhoushi, M., Shrivastava, A., Liskovich, D., Hosmer, B., Wasti, B., Lai, L., Mahmoud, A., Acun, B., Agarwal, S., Roman, A., Aly, A., Chen, B., & Wu, C.-J. (2024). LayerSkip: Enabling early exit inference and self-speculative decoding. Annual Meeting of the Association for Computational Linguistics (ACL).
- Fedus, W., Zoph, B., & Shazeer, N. (2022). Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. Journal of Machine Learning Research, 23(120), 1–39.
- Frantar, E., Ashkboos, S., Hoefler, T., & Alistarh, D. (2023). GPTQ: Accurate post-training quantization for generative pre-trained transformers. International Conference on Learning Representations (ICLR).
- Frantar, E., Castro, R. L., Chen, J., Hoefler, T., & Alistarh, D. (2024). MARLIN: Mixed-precision auto-regressive parallel inference on large language models. arXiv Preprint arXiv:2408.11743.
- Fu, Y., Bailis, P., Stoica, I., & Zhang, H. (2024). Break the sequential dependency of LLM inference using lookahead decoding. International Conference on Machine Learning (ICML).
- Gerganov, G. et al. (2023). Llama.cpp: LLM inference in C/C++. <https://github.com/ggml-org/llama.cpp>.
- He, Z., Zhong, Z., Cai, T., Lee, J. D., & He, D. (2024). REST: Retrieval-based speculative decoding. Proceedings of NAACL.
- Kwon, W. et al. (2023). Efficient memory management for large language model serving with PagedAttention. Proceedings of the 29th Symposium on Operating Systems Principles (SOSP).
- leejet, & contributors. (2025). Stable-diffusion.cpp. <https://github.com/leejet/stable-diffusion.cpp>.
- Leviathan, Y., Kalman, M., & Matias, Y. (2023). Fast inference from transformers via speculative decoding. International Conference on Machine Learning (ICML).
- Li, M., Lin, Y., Zhang, Z., Cai, T., Li, X., Guo, J., Xie, E., Meng, C., Zhu, J.-Y., & Han, S. (2025). SVDQuant: Absorbing outliers by low-rank components for 4-bit diffusion models. International Conference on Learning Representations (ICLR).
- Li, Y., Wei, F., Zhang, C., & Zhang, H. (2024). EAGLE: Speculative sampling requires rethinking feature uncertainty. International Conference on Machine Learning (ICML).
- Li, Y., Wei, F., Zhang, C., & Zhang, H. (2025). EAGLE-3: Scaling up inference acceleration of large language models via training-time test. arXiv Preprint arXiv:2503.01840.
- Lin, J. et al. (2024). AWQ: Activation-aware weight quantization for on-device LLM compression and acceleration. Proceedings of Machine Learning and Systems (MLSys).
- Liu, J., Ponnusamy, P., Cai, T., Guo, H., Kim, Y., & Athiwaratkun, B. (2025). Training-free activation sparsity in large language models. International Conference on Learning Representations (ICLR).

20. Liu, X., Daniel, C., Hu, L., Kwon, W., Li, Z., Mo, X., Cheung, A., Deng, Z., Stoica, I., & Zhang, H. (2024). Optimizing speculative decoding for serving large language models using goodput. arXiv Preprint arXiv:2406.14066.
21. Liu, Z. et al. (2023). Deja vu: Contextual sparsity for efficient LLMs at inference time. International Conference on Machine Learning (ICML).
22. Ollama. (2023). Ollama: Get up and running with large language models. <https://github.com/ollama/ollama>.
23. Pope, R. et al. (2023). Efficiently scaling transformer inference. Proceedings of Machine Learning and Systems (MLSys).
24. Saxena, A. (2023). Prompt lookup decoding.
25. SGLang Team. (2025). SpecForge: Training speculative decoding models for SGLang.
26. Shazeer, N. et al. (2017). Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. International Conference on Learning Representations (ICLR).
27. Shim, K., Lee, M., Choi, I., Boo, Y., & Sung, W. (2017). SVD-softmax: Fast softmax approximation on large vocabulary neural networks. Advances in Neural Information Processing Systems.
28. Tongyi Lab. (2025). Z-Image-Turbo. <https://huggingface.co/Tongyi-MAI/Z-Image-Turbo>.
29. Zheng, L. et al. (2024). SGLang: Efficient execution of structured language model programs. Advances in Neural Information Processing Systems (NeurIPS).