FASTKERNELS: BENCHMARKING GPU KERNEL GENERATION IN PRODUCTION

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM-based agents for GPU kernel generation are advancing rapidly, but the benchmarks they optimize against evaluate kernels in isolation, with synthetic inputs and weak baselines, rewarding sandbox speedups that break or vanish in real inference systems. We introduce FASTKERNELS, a benchmark of **384 tasks** drawn from 47 representative architectures across 8 categories, whose kernels suffice to reimplement **94.6% (472/499)** of HuggingFace Transformers architectures with outputs matching the native implementations. Each task mirrors the interface of the corresponding production module and is scored against the kernels production frameworks ship, and tasks form a compositional hierarchy, from primitives to full models, in which higher-level modules import lower-level ones. Candidates are scored at the kernel level and end to end inside the models they come from, on the production execution path, and MACROEVAL aggregates calibrated correctness, coverage, and speedup into a leaderboard. Seeding it with five representative agents (6,900 agent-hours), we find that kernel-level speedups of 1.6–$6.6\times$ shrink to **at most** $1.25\times$ end to end, **only 20%** of winning kernel sets run correctly as-is, and kernel-level scores **mis-rank agents**: Claude Code matches or beats KDA at every level in isolation, yet KDA scores $3\times$ higher end to end. Anonymized code is included in the supplementary material and will be open-sourced upon publication.

1 INTRODUCTION

LLM-based agents for GPU kernel generation now achieve strong scores on isolated benchmarks (Ouyang et al., 2025; Lin et al., 2026; Xing et al., 2026), with state-of-the-art systems autonomously compiling, profiling, and refining CUDA or Triton code. Yet these gains often fail to transfer into production inference frameworks such as vLLM (Kwon et al., 2023) and SGLang (Zheng et al., 2024): kernels that score well in sandbox evaluation routinely regress once they encounter real serving interfaces, compilation stacks, and workloads.

The root cause is **benchmark–production misalignment**. Current benchmarks rely heavily on synthetic inputs, single-GPU isolated kernels, simplified interfaces, and independent task levels that do not compose into full inference pipelines. They therefore reward kernels that are fast in isolation but brittle in deployed systems, hiding interface mismatches, compilation-stack conflicts, and correctness degradation that only appears at model scale.

OUR APPROACH

We introduce FASTKERNELS (Figure 1), a benchmark for kernel agents designed so that its scores predict deployment outcomes. FASTKERNELS derives 384 tasks top-down from 47 real model architectures across 8 categories; each task's interface matches the corresponding module in the production library for its family, and each candidate is scored against the kernels that library ships. Crucially, its task levels are compositional: Level 1 primitives feed Level 2 fused operators, which feed Level 3 layers and Level 4 models. Unlike benchmarks whose levels are independent, this hierarchy enables a dynamic-programming style optimization loop: an agent can reuse an optimized lower-level kernel, such as a linear operator, when optimizing higher-level modules such as MLPs or transformer blocks, instead of rediscovering the same building block from scratch.

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.

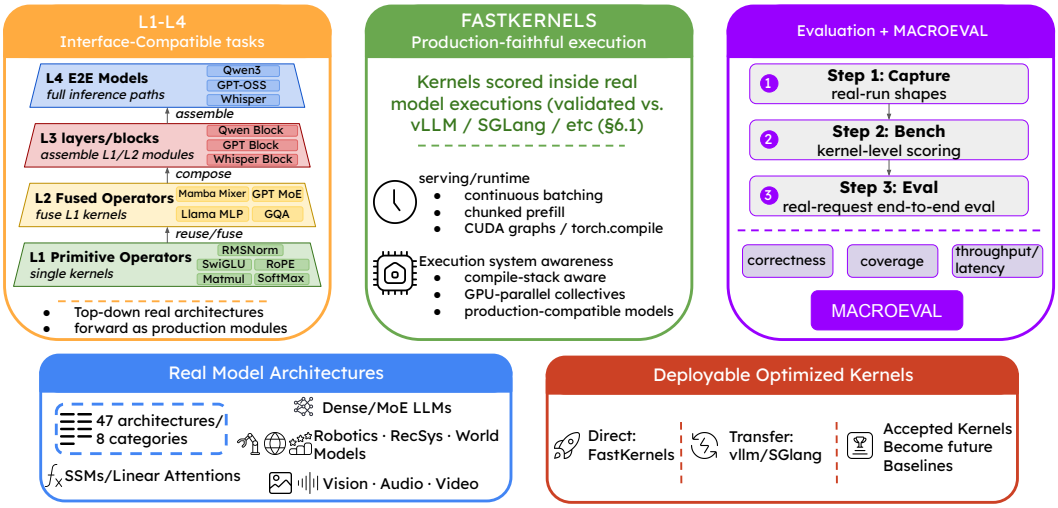


Figure 1: **FastKernels overview.** Tasks derived from 47 real architectures (bottom left) form a compositional L1–L4 hierarchy with production interfaces (top left) and run inside validated production model executions (center). Candidates are scored by capture, kernel-level bench, and end-to-end eval, aggregated by MacroEval (right); optimized kernels deploy directly or transfer into vLLM and SGLang (bottom right).

Because integration failures only surface at model scale, FastKernels also scores every candidate end to end, inside the models its tasks come from and on their production execution path (continuous batching, chunked prefill, CUDA graphs, `torch.compile`, tensor parallelism). This model runtime is the benchmark's measurement instrument, not a separate contribution, and we validate it against vLLM, SGLang, and the other production references before scoring any agent (Section 6.1). The result is an open leaderboard that ranks agents by what survives deployment, scored with MacroEval, and we seed it with a 6,900-agent-hour evaluation of five representative agents, from an RL-trained kernel model to contest-winning long-horizon harnesses.

Contributions

1. **Production-aligned task suite.** 384 tasks from 47 architectures across 8 categories, from dense and MoE LLMs and SSMs to vision, audio, robotics, 3D, recommendation, and world models, with production module interfaces and speedups measured against production kernels.
2. **Compositional task hierarchy.** Unlike flat benchmarks, the primitives-to-models hierarchy exposes *composition effects*: 37% of the agents' L3 winners break once their own lower-level kernels are active, and regenerating them on frozen lower-level winners yields 80 correct L3 winners instead of 45 (Section 6.2).
3. **Production-faithful evaluation.** Kernel-level scoring on captured production shapes, with collectives on real tensor-parallel groups, and end-to-end scoring inside full models.
4. **A deployment-grounded leaderboard.** MacroEval ranks agents by calibrated correctness, coverage, and end-to-end speedup across model families. Our seed evaluation of five representative agents shows kernel-level speedups of up to 6.6× shrinking to at most 1.25× end to end, and kernel-level scores that mis-rank agents (Section 6.3; Appendix L).

2 Related Work

GPU kernel benchmarks. KernelBench (Ouyang et al., 2025) pioneered the evaluation of LLM-generated GPU kernels, drawing from an earlier generation of architectures (e.g., AlexNet, VGG). Robust-kbench (Lange et al., 2025) addressed KernelBench's numerical instabilities and reward-hacking vulnerabilities. SOL-ExecBench (Lin et al., 2026) scores subgraphs against B200 speed-of-light bounds rather than software baselines. FlashInfer-Bench (Xing et al., 2026) integrates with

an inference engine but is limited to FlashInfer LLM operators, while CUDABENCH (Zhu et al., 2026) and TRITONBENCH (Li et al., 2025a) evaluate operators in isolation. Further efforts (Saroufim et al., 2025; Wen et al., 2025; Nie et al., 2026; Corporation, 2025; Kalade & Schelle, 2025) broaden backend and robustness coverage while retaining an operator-level, single-GPU scope. Concurrent benchmarks target multi-GPU code directly: COMMBENCH (Ma et al., 2026) evaluates communication programs across NVLink and RDMA, and PARALLELKERNELBENCH (Chan et al., 2026) asks agents to replace PyTorch+NCCL references with fused multi-GPU kernels; both score each program in isolation. FASTKERNELS is complementary and fills four gaps: it is the first kernel benchmark to (i) organize tasks into a compositional hierarchy from primitives to full models, mirroring production assembly, and to show that this hierarchy produces measurable composition effects (Section 6.2). It also (ii) measures speedup against the kernels shipped in state-of-the-art inference frameworks rather than PyTorch or theoretical bounds, (iii) matches production module interfaces, enabling copy–paste deployment, and (iv) scores communication kernels inside the tensor-parallel models that issue them.

LLM-based kernel agents. A parallel line develops inference-time agents combining LLM reasoning with profiler feedback or evolutionary search (Lange et al., 2025; Cheng et al., 2026; Liao et al., 2026; Wei et al., 2025; Zhang et al., 2025; Wang et al., 2025; Novikov et al., 2025; Sharma, 2025; Zou et al., 2026; Xie et al., 2026), alongside specialized models trained via RL or fine-tuning (Baronio et al., 2025; Li et al., 2026; Fisches et al., 2025; Li et al., 2025b; Zhu et al., 2025). These systems are typically developed against operator-level benchmarks, which FASTKERNELS complements.

3 BENCHMARK DESIGN

FASTKERNELS derives all tasks top-down from real model architectures, ensuring that every kernel can be traced to a specific layer in a specific model and evaluated end-to-end.

3.1 TOP-DOWN, MODEL-DRIVEN CONSTRUCTION

Unlike prior benchmarks that construct tasks bottom-up by combining primitive operators (Ouyang et al., 2025) or extracting subgraphs via automated pipelines (Lin et al., 2026), FASTKERNELS takes a *top-down* approach. We begin with real model families—selected to represent the current and near-future frontier of AI workloads—and recursively decompose them into the kernels that constitute their inference paths.

Task construction. For each model architecture, we load the HuggingFace configuration and architecture definition, walk the forward pass, and produce standalone task implementations for each computational kernel with all configuration constants (hidden size, number of heads, data types) inlined from the model's configuration. Every task is then audited to verify: (i) semantic correctness of the reference implementation, (ii) that the task captures the operator as it actually executes in the model (not a simplified proxy), (iii) that tensor shapes and data types match the model's real configuration, and (iv) that the task's interface matches the corresponding module in the production reference library (Section 3.4). The construction pipeline ships with FASTKERNELS, so users can add new architectures from HuggingFace with minimal effort. No task is synthetic: unlike KernelBench's Level-2 random fusions or SOL-ExecBench's LLM-selected subgraphs, every task is an operation that runs in a real model during inference.

3.2 TASK HIERARCHY

FASTKERNELS organizes tasks into four levels of increasing scope:

- **Level 1 — Primitive operators.** Individual operations: attention variants (GQA, MLA, sliding-window), normalizations (RMSNorm, LayerNorm), activations (SwiGLU, GeGLU), positional encodings (RoPE, ALiBi), and quantization/dequantization routines.
- **Level 2 — Fused operators.** Multi-operation kernels representing natural fusion opportunities as they arise in real models: residual-add + RMSNorm + quantization, attention + output projection, MoE gate + dispatch + expert computation.

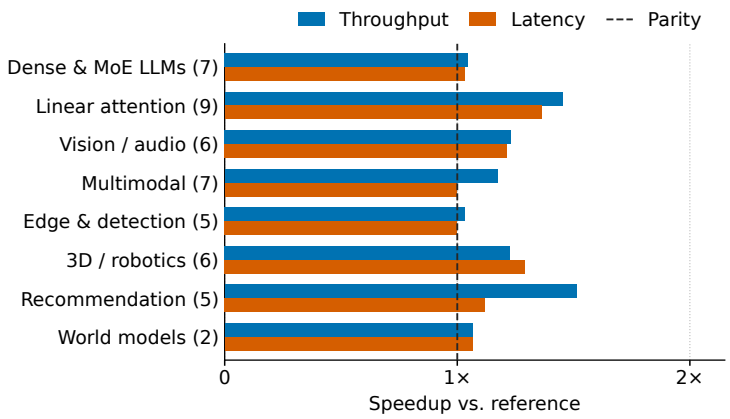


Figure 2: **Composition and reference performance of FASTKERNELS**: 47 end-to-end architectures across 8 categories (count in parentheses). Each bar is the geometric-mean end-to-end speedup of FASTKERNELS over the production reference across the category's architectures (throughput and latency). Per-architecture details in Appendix Tables 3, 5, and 6.

- **Level 3 — Full layers and blocks.** Complete architectural blocks: transformer decoder layers, SSM scan blocks, MoE layers with routing, cross-attention layers, and vision encoder stages.
- **Level 4 — End-to-end model architectures.** Full model inference paths evaluated as integrated systems. The 47 representative architectures in Level 4 serve as the end-to-end evaluation set, but the underlying L1–L3 kernels cover most operators across ML models.

Standalone and composed evaluation. Crucially, the hierarchy is not just organizational—it is *executable*. L2–L4 baselines import their lower-level dependencies via relative imports (e.g., an L3 decoder layer does `from ..L2.attention import Attention`), mirroring how production frameworks assemble modules. In *standalone* mode (`--standalone`), each candidate is benched in isolation: imported lower-level modules resolve to their production baselines regardless of what other candidate files exist. In *composed* mode, all candidate files in the candidate directory are active simultaneously, so an L3 block's lower-level imports resolve to agent-generated candidates rather than production baselines. The standalone score measures how well the agent optimized one module; the composed score measures what a real deployment would observe when multiple agent-generated modules interact. The gap between the two—the *composition effect*—is a first-order signal that flat benchmarks are structurally unable to produce (Section 6.2).

3.3 ARCHITECTURE COVERAGE

FASTKERNELS is designed to cover as many model architectures as possible with the *minimum* number of kernels, by consolidating near-identical operators (e.g., RoPE variants, normalizations, attention backends) into a single L1/L2 task and picking a small set of representative architectures whose union of operators subsumes the long tail of model families. The resulting 47 architectures span 8 categories and precision formats from 1.58-bit through FP32, and comprise 384 unique tasks: 108 L1, 155 L2, and 74 L3 kernel tasks, shared across architectures, plus the 47 end-to-end L4 models (Figure 2; counting rules and per-architecture details in Appendix A).

To measure coverage, we implemented every architecture in HuggingFace Transformers (commit `89b6b17`) on FASTKERNELS without adding any operator: **94.6% (472/499)** of them run and match native HF outputs within `dtype`-dependent tolerances, and only 5 require a genuinely new operation. The methodology and per-architecture results are in Appendix M.

3.4 INTERFACE-COMPATIBLE DESIGN

A central design principle of FASTKERNELS is that optimized kernels should be deployable into existing production systems with minimal effort. For each model architecture family, we identify

the corresponding state-of-the-art production library (e.g., vLLM for LLMs, SGLang for serving) and design each task's interface—its `__init__` constructor signature and `forward` method—to closely match the corresponding module in that reference library.

A kernel optimized within FASTKERNELS can therefore be copied into vLLM, SGLang, or another production framework without interface refactoring. Unlike FlashInfer-Bench's FIApply, which substitutes at the kernel dispatch level (an abstraction internal to FlashInfer), FASTKERNELS's compatibility operates at the *module* level, the unit production frameworks compose.

3.5 MULTI-GPU COMMUNICATION KERNELS

Production inference at scale is almost always distributed, and its collectives and communication–computation overlaps materially affect end-to-end latency. FASTKERNELS includes the communication path of its tensor-parallel models as tasks: the custom IPC all-reduce (L1); the tensor-parallel linear, embedding, and MoE layers that issue it (L2); and a decoder layer that fuses it with residual addition and RMSNorm (L3). The all-reduce and row-parallel linear are benched in a real NCCL group of the capture's TP degree, and all of these operators run end to end inside models served at TP 2–8. Dedicated benchmarks (Ma et al., 2026; Chan et al., 2026) cover more communication patterns but score each program in isolation; FASTKERNELS places every collective in the schedule it shares with the model's compute kernels, where a kernel $1.5\times$ faster on one GPU can still slow the model.

4 BENCHMARKING STACK

FASTKERNELS exposes one evaluation stack through two tiers, *kernel-level* (`fastkernels bench`) and *end-to-end* (`fastkernels eval`), each usable as a diagnostic tool or in leaderboard mode.

Scenario sets. A scenario fixes a checkpoint, TP degree, dtype, and workloads from public datasets for its modality (e.g., WildChat for LLMs, Parti-prompts for diffusion, COCO for detection). The *full* set has one scenario per architecture (47 scenarios, 184 workloads, TP 1–8) and is used to validate the benchmark (Section 6.1). The *default* set is the 11-model subset on which agents are scored, because a single long-horizon agent already spends up to 2,500 agent-hours on it: it spans all 8 categories, TP 1–4, and BF16/FP16/FP8/MXFP4 (48 workloads), and its models run 133 distinct L1–L3 tasks (Appendix C).

Workload capture. `fastkernels capture` runs each scenario end to end and records, for every task module, a reconstructable constructor recipe and the shape, dtype, and stride of every `forward` argument. Identical calls merge into one variant with a call count; the kernel tier then merges variants across workloads and benchmarks up to five per operator (smallest, median, and largest problem, then the most frequent): on the default B200 capture, 12.4M calls reduce to 87.7K distinct cases and 582 benchmarked ones (Appendix D).

Kernel-level tier. The kernel tier rebuilds each candidate next to its baseline from the recipe, shares weights, and compares outputs and runtimes on the selected cases, one operator per isolated process with reward-hack checks. Values are synthesized from the captured shapes, with offsets, indices, and cache slots kept consistent with the call; for data-dependent operators such as MoE dispatch, FASTKERNELS additionally replays golden inputs captured from real end-to-end executions, ensuring that both correctness and performance are measured on the actual tensors encountered by the model (Appendix E).

End-to-end tier. The end-to-end tier swaps a candidate set into full model executions on the production path and real requests, next to a baseline run and a noise run that calibrates correctness thresholds, and produces the MACROEVAL metrics (Section 5). The kernel tier gives agents fast feedback; the end-to-end tier is the ground truth that the leaderboard reports. Both tiers integrate with profilers, NVIDIA Nsight Compute (NCU) for kernel-level analysis and Nsight Systems (NSYS) for end-to-end traces, with LLM-based extraction of the main bottlenecks, and with MLflow for tracking kernel lineage across agent runs.

5 MACROEVAL: CROSS-ARCHITECTURE METRICS

FASTKERNELS evaluates candidate kernels by substituting them into end-to-end model executions and comparing each run with a reference run that uses the production baseline kernels. The key challenge is that model families expose different correctness signals—tokens, embeddings, labels, rankings, trajectories, audio, or video—and different throughput–latency tradeoffs. MACROEVAL addresses this by calibrating architecture-specific correctness to a common $[0, 1]$ scale, measuring end-to-end speedup, and macro-averaging across families so no single architecture dominates the leaderboard.

Benchmark indexing. Let $f \in \mathcal{F}$ index architecture families, such as LLMs, diffusion models, vision models, speech and audio models, retrieval and recommender models, and robotics or world models. For family f, let $i \in B_f$ index benchmark items, and let $r \in R_i$ index requests, prompts, seeds, or input instances for item i. We write $y_{\mathrm{ref},i,r}$ and $y_{\mathrm{cand},i,r}$ for the reference and candidate outputs, and $T_{\mathrm{ref},i}$ and $T_{\mathrm{cand},i}$ for their measured runtimes.

Calibrated correctness. Raw output discrepancies are inherently architecture-dependent: teacher-forced top-1 disagreement with the reference tokens for autoregressive models (free-running greedy outputs diverge even between two correct runs, e.g. after a fresh compile), embedding or output distance for encoders and diffusion models, IoU discrepancy for detection or segmentation, WER/CER deltas for speech, ranking disagreement for retrieval, and trajectory error for robotics. For each request, FASTKERNELS therefore computes a discrepancy $d_{i,r} = D_f(y_{\mathrm{cand},i,r}, y_{\mathrm{ref},i,r})$, with D_f chosen for the output type of family f, and maps it to a calibrated correctness score in $[0, 1]$:

$$C_{i,r} = \begin{cases} 1, & d_{i,r} \le g_i, \\ \frac{f_i - d_{i,r}}{f_i - g_i}, & g_i < d_{i,r} < f_i, \\ 0, & d_{i,r} \ge f_i. \end{cases} \tag{1}$$

Here g_i is the threshold below which the candidate is indistinguishable from the reference, and f_i is the threshold above which the output is incorrect or unusable. Thresholds are fixed before evaluation. g_i is calibrated per model from reference-vs-reference nondeterminism: a *noise run* repeats the production reference in a fresh process with a cold compile cache (re-making every compilation and autotuning choice, as a candidate run does), and g_i is the 95th percentile of its per-sample discrepancy against the reference (with a floor of 10^{-3}; OpenFold3, whose noise run is bit-identical, is calibrated with a benign numerical perturbation instead, Appendix K). We set $f_i = 0.5$ for every family—a candidate at least half as far from the reference as an unrelated output receives no credit—and $\tau_i = 0.9$; Appendix K shows that the ranking is unchanged when g_i and f_i are halved or doubled.

Correctness is averaged over requests, then over items within each family, then over families:

$$C_i = \frac{1}{|R_i|} \sum_{r \in R_i} C_{i,r}, \qquad C_f = \frac{1}{|B_f|} \sum_{i \in B_f} C_i, \qquad C_{\mathrm{macro}} = \frac{1}{|\mathcal{F}|} \sum_{f \in \mathcal{F}} C_f. \tag{2}$$

This macro-family weighting gives each architecture family equal influence and prevents overrepresented families, such as LLMs, from dominating the correctness axis.

Validity and coverage. Although calibrated correctness is continuous, deployment requires a hard validity decision for each benchmark item. We define $\mathrm{valid}_i = \mathbf{1}(C_i \ge \tau_i)$ for a pre-specified item or family threshold τ_i. Items are invalid if they crash, hang, trigger shape or type errors, access memory illegally, produce NaNs, or fall below the correctness threshold. We report both item-level coverage and macro-family coverage:

$$\mathrm{Coverage} = \frac{1}{N} \sum_i \mathrm{valid}_i, \qquad \mathrm{Coverage}_{\mathrm{macro}} = \frac{1}{|\mathcal{F}|} \sum_{f \in \mathcal{F}} \frac{1}{|B_f|} \sum_{i \in B_f} \mathrm{valid}_i \,. \tag{3}$$

Macro-coverage is the preferred statistic when architecture balance matters.

Throughput–latency speedup. For valid items (invalid items receive no speedup credit), performance is measured against the production reference with a geometric blend of throughput and latency

speedups:

$$s_i = \left(s_i^{\text{thr}}\right)^{\lambda} \left(s_i^{\text{lat}}\right)^{1-\lambda}, \qquad s_i^{\text{thr}} = \frac{\text{Throughput}_{\text{cand},i}}{\text{Throughput}_{\text{ref},i}}, \qquad s_i^{\text{lat}} = \frac{\text{Latency}_{\text{ref},i}}{\text{Latency}_{\text{cand},i}}. \tag{4}$$

The default $\lambda = 0.5$ is neutral: an optimization that improves throughput while equivalently degrading latency receives no net credit, since such gains are workload-dependent tradeoffs rather than universal improvements; users may pre-specify a different λ. Because speedups are multiplicative, FASTKERNELS aggregates the blended speedups with geometric means. For each family, let $B_f^{\text{valid}} = \{i \in B_f : \text{valid}_i = 1\}$. The family-level and macro-geometric speedups are

$$S_f = \exp\left(\frac{1}{|B_f^{\text{valid}}|} \sum_{i \in B_f^{\text{valid}}} \log s_i\right), \qquad S_{\text{macro}} = \exp\left(\frac{1}{|\mathcal{F}_{\text{valid}}|} \sum_{f \in \mathcal{F}_{\text{valid}}} \log S_f\right), \tag{5}$$

where $\mathcal{F}_{\text{valid}}$ contains families with at least one valid item. A family with no valid item is excluded from speedup aggregation, so incorrect kernels earn no performance credit, and the failure is penalized through macro-coverage instead.

Leaderboard. The leaderboard reports S_{macro}, C_{macro}, and $\text{Coverage}_{\text{macro}}$ separately, so users can rank agents by whichever their deployment prioritizes. As the default single ranking, FASTKERNELS uses their product $\text{Score}_{\text{default}} = S_{\text{macro}} \cdot C_{\text{macro}} \cdot \text{Coverage}_{\text{macro}}$, which rewards agents that are simultaneously fast, correct, and broadly applicable, and it also reports throughput-only (λ=1) and latency-only (λ=0) rankings for batched and interactive deployments. All speedups are measured empirically against the reference, without speed-of-light or roofline bounds.

6 EVALUATION

The evaluation has two parts: Section 6.1 evaluates the *benchmark itself*, checking that its baselines match each family's production framework so that speedups over them are meaningful; Sections 6.2 and 6.3 evaluate *kernel agents* on the default set, at the kernel level and end to end.

6.1 BENCHMARK VALIDATION: BASELINES VS. PRODUCTION FRAMEWORKS

On the full set, we benchmark every architecture's end-to-end performance (throughput and latency on real workloads) and the correctness of each L1–L4 task against the state-of-the-art production framework for its family (e.g., vLLM for LLMs, SGLang for serving, diffusers for SDXL, FLA for linear-attention recurrences, timm for vision encoders). Figure 2 reports the category-level geometric means: gains concentrate in linear attention ($1.45\times$ throughput) and recommendation ($1.51\times$), and the other categories sit near parity; Appendix G breaks the results down by architecture and GPU (Blackwell vs. Hopper). References are pinned per release and GPU (vLLM 0.26.0 here); a daily CI job tracks their drift, 1–9% for the median model over the four months before our pin (Appendix H).

6.2 AGENT EVALUATION: KERNEL-LEVEL RESULTS

Agents. We evaluate Dr. Kernel (Liu et al., 2026), an RL-trained model and one of the most recent KernelBench-ecosystem agents (others did not reproduce their results in our setting, e.g. KernelAgent (Cheng et al., 2026), or assume KernelBench's single-file tasks); KDA (Zou et al., 2026) and AKO (Xie et al., 2026) (adapted by the GEMM People entry), long-horizon harnesses behind winning MLSys 2026 FlashInfer contest entries (FlashInfer Team, 2026): KDA placed in all three full-agent tracks and GEMM People second on MoE; and Claude Code and Codex (OpenAI, 2025), the coding agents these harnesses build on. Their campaigns total 6,900 agent-hours and 42B tokens, with Claude Code, KDA, and AKO spending a median of 6–12 hours per operator (Appendix I). We therefore evaluate fewer agents than single-turn leaderboards: the agents practitioners deploy iterate for hours per kernel (contest teams had 13 weeks for five kernels). We omit evolutionary search, which multiplies this cost by population size and generation count and whose strongest systems (Novikov et al., 2025; Liao et al., 2026) are not public. Each agent optimizes every L1–L3 operator captured from the default set on B200 (47 L1, 64 L2, and 21 L3 targets) against the human-expert kernels that production frameworks ship. L4 tasks are full models, so benchmarking

Table 1: **Kernel agents on FASTKERNELS (B200, default set).** *Corr.*: operators correct on every benchmarked case; *Geo.*: geomean speedup over correct operators; fast_p: correct and more than $p\times$ faster than production (Appendix J). Composed and sequential rows contain winners only.

		L1 (47 targets)				**L2** (64 targets)				**L3** (21 targets)			
Agent	Mode	Corr.	Geo.	fast_1	fast_2	Corr.	Geo.	fast_1	fast_2	Corr.	Geo.	fast_1	fast_2
Dr. Kernel	Independent	31	0.93×	18	1	18	0.59×	5	0	4	0.97×	1	0
	Composed	18	1.29×	18	1	5	1.07×	4	0	1	1.03×	1	0
	Sequential	18	1.29×	18	1	5	1.40×	5	0	2	1.16×	2	0
Claude Code	Independent	47	1.81×	42	10	64	2.66×	56	37	20	4.74×	19	16
	Composed	42	1.79×	40	10	54	3.52×	54	36	14	6.28×	14	12
	Sequential	42	1.76×	40	10	57	3.21×	57	37	20	4.50×	20	15
KDA	Independent	47	1.64×	42	8	63	2.56×	61	37	20	3.05×	18	14
	Composed	43	1.69×	40	9	51	2.62×	51	31	11	3.45×	11	9
	Sequential	43	1.68×	40	8	57	2.30×	57	30	20	2.86×	20	14
AKO	Independent	45	1.84×	44	11	64	3.35×	64	41	18	6.48×	18	17
	Composed	45	1.87×	44	11	60	3.54×	60	43	14	6.59×	14	13
	Sequential	45	1.86×	44	11	59	3.51×	59	40	20	6.55×	20	17
Codex	Independent	47	1.62×	40	9	64	2.87×	60	39	19	2.92×	18	12
	Composed	41	1.76×	40	9	47	3.52×	45	33	5	5.79×	5	4
	Sequential	41	1.75×	38	9	56	2.90×	56	34	18	3.95×	18	11

them is the end-to-end evaluation of Section 6.3. We bench L1–L3 in three modes: *independent* (each operator generated and benched in isolation, over production lower-level kernels), *composed* (the independent winners—correct, geomean speedup $\geq 1\times$—activated together, so higher-level kernels run on the agent's own lower-level winners), and *sequential* (L2 and L3 regenerated on top of the frozen lower-level winners, benched composed). Four agents reach geomean speedups of 1.6–6.6× over the production kernels (Table 1), but composition erodes them (Appendix Table 12): once their children are also agent-generated, 26 of the agents' 71 L3 winners (37%) fail correctness (13 of 18 for Codex), the surviving L3 winners of KDA and AKO lose 12–14% of their speedup, and individual modules lose up to 76% (Claude Code's `yolov10_backbone`, $30.3\times \rightarrow 7.2\times$). Sequential generation makes this an optimization target: it yields 80 correct L3 winners across agents instead of 45 composed, and the sequential L3 kernels of AKO and Codex are $1.48\times$ and $1.35\times$ faster than their composed independent ones.

6.3 AGENT EVALUATION: END-TO-END LEADERBOARD

We deploy each agent's independent and sequential winner sets into the 11 default models (each receiving the kernels it instantiates) and run their real workloads on the production path, next to a baseline run and a noise run (Section 5). If a set crashes the model or grossly corrupts its outputs, the offending kernel (named in the traceback, else found by bisection) is dropped and the model re-run; the first subset that runs correctly is the agent's *deployable subset*, timed on the full workloads. Table 2 scores both the full winner sets as-is (*all winners*) and the deployable subsets (per model: Appendix K).

Kernel-level winners rarely survive deployment as-is. Swapped in unchanged, only 22 of the 110 winner-set deployments run correctly (13–58% of model families per agent). Compilation-stack conflicts (e.g. embedding kernels calling raw C++ extensions that `torch.compile` cannot trace, rotary kernels that specialize dynamic batch shapes) and crashes on engine shapes the kernel bench never produced account for 86% of the drops with an identified cause (Appendix Table 13), all invisible to an isolated kernel check. Other sets pass every kernel check yet corrupt the model (texture-only FLUX images, NaNs on Oasis), which only the end-to-end check flags.

Deployable end-to-end gains are small. Even after dropping the failing kernels, end-to-end speedups are modest (S_{macro} of 0.96–1.25×) despite kernel-level speedups of up to 6.6×: gains concentrate where the rewritten kernels dominate runtime (up to 2.2× on YOLOv10, 1.7× on

Table 2: **End-to-end MACROEVAL leaderboard on the default set (B200).** *Rank*: position of each agent–mode entry by Score (10 entries). *Valid*: models with $C_i \geq \tau$; macro metrics as in Section 5; *Thr./Lat.*: λ=1/λ=0. *Deployable subset*: after dropping the kernels that crash or break the model; *All winners*: every kernel-level winner as-is.

		Deployable subset								All winners	
					S_{macro}						
Agent	Mode	Valid	$\text{Cov}_{\text{macro}}$	C_{macro}	λ=0.5	Thr.	Lat.	Score	Rank	Valid	$\text{Cov}_{\text{macro}}$
Dr. Kernel	Independent	9/11	0.75	0.85	0.98×	1.01×	0.95×	0.63	4	2/11	0.25
	Sequential	9/11	0.75	0.75	0.99×	1.02×	0.96×	0.55	5	2/11	0.25
Claude Code	Independent	6/11	0.52	0.61	1.09×	1.08×	1.11×	0.35	8	2/11	0.25
	Sequential	7/11	0.56	0.55	1.11×	1.09×	1.14×	0.34	9	1/11	0.12
KDA	Independent	11/11	1.00	0.98	1.13×	1.08×	1.19×	**1.11**	**1**	4/11	0.42
	Sequential	11/11	1.00	0.98	1.04×	1.03×	1.05×	1.03	2	6/11	0.58
AKO	Independent	7/11	0.56	0.65	1.15×	1.13×	1.16×	0.42	7	1/11	0.12
	Sequential	9/11	0.83	0.82	1.25×	1.08×	1.44×	0.85	3	2/11	0.25
Codex	Independent	5/11	0.33	0.33	0.96×	0.98×	0.94×	0.11	10	1/11	0.12
	Sequential	8/11	0.71	0.71	1.10×	1.11×	1.08×	0.55	6	1/11	0.12

OpenFold3, $1.5\times$ on GLA), while LLM serving stays within 6% of the production kernels. Sequential sets raise AKO and Codex from $1.15\times$ and $0.96\times$ to $1.25\times$ and $1.10\times$.

Kernel-level scores mis-rank agents. In isolation, Claude Code matches or beats KDA on correctness and speedup at every level (L3: 20 vs. 20 correct, $4.74\times$ vs. $3.05\times$), yet KDA's MACROEVAL score is $3\times$ higher (1.11 vs. 0.35), the best of all agents even with halved or doubled thresholds (Appendix K). The gap is deployability, not speed ($1.13\times$ vs. $1.09\times$): all 11 of KDA's models are valid versus 6 of Claude Code's. KDA wraps the same Claude model in a planning and review harness, which a kernel-level leaderboard cannot credit.

7 CONCLUSION

We presented FASTKERNELS, a benchmark that grounds kernel-generation evaluation in production inference. Our seed evaluation shows that kernel-level success barely predicts deployment: speedups of up to $6.6\times$ shrink to at most $1.25\times$ end to end, only 20% of winning kernel sets run correctly as-is, and kernel-level scores rank Claude Code above KDA, which scores $3\times$ higher.

Limitations. The 94.6% HuggingFace coverage is checked on a few forward passes per architecture, some with randomly initialized weights, so it certifies functional equivalence rather than task accuracy. Performance is measured on locked-clock B200 SXM; rankings on other GPUs may differ. Our kernel-level results use synthesized inputs, so data-dependent kernels are scored faithfully only end to end; agents are scored on the default set only. MACROEVAL's family weighting, λ=0.5 blend, and product score encode value judgments, so we also report its components.

AI USE STATEMENT

LLM-based kernel-generation agents are a core experimental subject of this work: we evaluate Dr. Kernel, Claude Code, KDA, AKO, and Codex as described in Section 6.2 and Section 6.3. We used generative AI tools to edit software associated with the benchmark, to draft and refine portions of the manuscript, and to assist with table, figure, and reference formatting. We have not used generative AI tools to generate synthetic datasets, formulate mathematical claims, or develop theoretical models; those tasks are not applicable to this work. All AI-assisted text, code, figures, and experimental configurations were reviewed by the authors. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This paper introduces a benchmark and evaluation methodology for GPU kernel generation. It does not involve human subjects, crowdsourcing, or the release of a new high-risk generative model. Tasks are derived from publicly available model architectures, libraries, and datasets. The main dual-use consideration is that more efficient kernels can lower the cost of deploying large models; we mitigate silent failures by requiring production-faithful correctness checks before speedup credit is awarded. The authors have read and adhere to the ICLR Code of Ethics.

REPRODUCIBILITY STATEMENT

An anonymized implementation of FASTKERNELS and the associated evaluation scripts is included in the supplementary material and will be open-sourced upon publication. Section 3 describes task construction, the compositional hierarchy, and production-matched interfaces; Section 4 describes the evaluation stack; and Section 5 defines MACROEVAL. Scenario sets, the capture and case-selection pipeline, and agent configurations and costs are documented in Appendices C, D, and I; hardware, serving workloads, threshold calibration, sensitivity analyses, and per-architecture coverage in Appendix F, Appendix K, and Appendix G.

REFERENCES

Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. Kevin: Multi-turn rl for generating cuda kernels, 2025. URL https://arxiv.org/abs/2507.11948.

Willy Chan, Nathan Paek, Simon Guo, Simran Arora, and Daniel Fu. ParallelKernelBench: Can LLMs write fast multi-GPU kernels? https://github.com/togethercomputer/ParallelKernelBench, 2026.

Kaiming Cheng, Laura Wang, Jack Khuu, Mark Saroufim, Wenyuan Chi, Jiannan Wang, and Joe Isaacson. KernelAgent: Hardware-guided GPU kernel optimization via multi-agent orchestration. PyTorch Blog, March 2026. URL https://pytorch.org/blog/kernelagent-hardware-guided-gpu-kernel-optimization-via-multi-agent-orchestration/. Accessed: 2026-05-06.

NVIDIA Corporation. Computeeval: A benchmark for evaluating large language models on cuda code generation, 2025. URL https://github.com/NVIDIA/compute-eval.

Zacharias V. Fisches, Sahan Paliskara, Simon Guo, Alex Zhang, Joe Spisak, Chris Cummins, Hugh Leather, Gabriel Synnaeve, Joe Isaacson, Aram Markosyan, and Mark Saroufim. Kernelllm: Making kernel development more accessible, 6 2025. URL https://huggingface.co/facebook/KernelLLM.

FlashInfer Team. MLSys 2026 FlashInfer AI kernel generation contest. https://mlsys26.flashinfer.ai/, 2026.

Sarunas Kalade and Graham Schelle. Npueval: Optimizing npu kernels with llms and open source compilers, 2025. URL https://arxiv.org/abs/2507.14403.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

Robert Tjarko Lange, Qi Sun, Aaditya Prasad, Maxence Faldor, Yujin Tang, and David Ha. Towards robust agentic cuda kernel benchmarking, verification, and optimization, 2025. URL https://arxiv.org/abs/2509.14279.

Jianling Li, Shangzhan Li, Zhenye Gao, Qi Shi, Yuxuan Li, Zefan Wang, Jiacheng Huang, Haojie Wang, Jianrong Wang, Xu Han, Zhiyuan Liu, and Maosong Sun. Tritonbench: Benchmarking large language model capabilities for generating triton operators, 2025a. URL https://arxiv.org/abs/2502.14752.

Shangzhan Li, Zefan Wang, Ye He, Yuxuan Li, Qi Shi, Jianling Li, Yonggang Hu, Wanxiang Che, Xu Han, Zhiyuan Liu, and Maosong Sun. Autotriton: Automatic triton programming with reinforcement learning in llms, 2025b. URL https://arxiv.org/abs/2507.05687.

Xiaoya Li, Xiaofei Sun, Albert Wang, Jiwei Li, and Chris Shum. Cuda-l1: Improving cuda optimization via contrastive reinforcement learning, 2026. URL https://arxiv.org/abs/2507.14111.

Gang Liao, Hongsen Qin, Ying Wang, Alicia Golden, Michael Kuchnik, Yavuz Yetim, Jia Jiunn Ang, Chunli Fu, Yihan He, Samuel Hsia, Zewei Jiang, Dianshi Li, Uladzimir Pashkevich, Varna Puvvada, Feng Shi, Matt Steiner, Ruichao Xiao, Nathan Yan, Xiayu Yu, Zhou Fang, Roman Levenstein, Kunming Ho, Haishan Zhu, Alec Hammond, Richard Li, Ajit Mathews, Kaustubh Gondkar, Abdul Zainul-Abedin, Ketan Singh, Hongtao Yu, Wenyuan Chi, Barney Huang, Sean Zhang, Noah Weller, Zach Marine, Wyatt Cook, Carole-Jean Wu, and Gaoxiang Liu. Kernelevolve: Scaling agentic kernel coding for heterogeneous ai accelerators at meta, 2026. URL https://arxiv.org/abs/2512.23236.

Edward Lin, Sahil Modi, Siva Kumar Sastry Hari, Qijing Huang, Zhifan Ye, Nestor Qin, Fengzhe Zhou, Yuan Zhang, Jingquan Wang, Sana Damani, Dheeraj Peri, Ouye Xie, Aditya Kane, Moshe Maor, Michael Behar, Triston Cao, Rishabh Mehta, Vartika Singh, Vikram Sharma Mailthody, Terry Chen, Zihao Ye, Hanfeng Chen, Tianqi Chen, Vinod Grover, Wei Chen, Wei Liu, Eric Chung, Luis Ceze, Roger Bringmann, Cyril Zeller, Michael Lightstone, Christos Kozyrakis, and Humphrey Shi. Sol-execbench: Speed-of-light benchmarking for real-world gpu kernels against hardware limits, 2026. URL https://arxiv.org/abs/2603.19173.

Wei Liu, Jiawei Xu, Yingru Li, Longtao Zheng, Tianjian Li, Qian Liu, and Junxian He. Dr. kernel: Reinforcement learning done right for triton kernel generations, 2026. URL https://arxiv.org/abs/2602.05885.

Shuang Ma, Yuyi Li, Yihan Zhang, Hezhi Xie, Danyang Chen, Shuyang Ji, Ziming Mao, Cheng Ji, Ansha Prashanth, Wenting Yang, Yiran Wang, Chihan Cui, Pei Yu Lin, Ion Stoica, and Yang Zhou. CommBench: Can LLMs write correct and efficient GPU communication code?, 2026. URL https://arxiv.org/abs/2608.04450.

Jiayi Nie, Haoran Wu, Yao Lai, Zeyu Cao, Cheng Zhang, Binglei Lou, Erwei Wang, Jianyi Cheng, Timothy M. Jones, Robert Mullins, Rika Antonova, and Yiren Zhao. Kernelcraft: Benchmarking for agentic close-to-metal kernel generation on emerging hardware, 2026. URL https://arxiv.org/abs/2603.08721.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. Alphaevolve: A coding agent for scientific and algorithmic discovery, 2025. URL https://arxiv.org/abs/2506.13131.

OpenAI. OpenAI Codex. https://openai.com/index/introducing-codex/, 2025. Accessed: 2026-05-06.

Anne Ouyang, Simon Zheng, Ce Gao, Yida Dong, Kunhao He, Lisa Li, Saurav Nrusimha, Daniel Zhu, Fei Song, et al. Kernelbench: Can LLMs write efficient gpu kernels? In *Proceedings of the International Conference on Machine Learning (ICML)*, 2025.

Mark Saroufim, Jiannan Wang, Bert Maher, Sahan Paliskara, Laura Wang, Shahin Sefati, and Manuel Candales. Backendbench: An evaluation suite for testing how well llms and humans can write pytorch backends, 2025. URL https://github.com/meta-pytorch/BackendBench.

Asankhaya Sharma. OpenEvolve: An open-source evolutionary coding agent, 2025. URL https://github.com/codelion/openevolve.

Jianghui Wang, Vinay Joshi, Saptarshi Majumder, Xu Chao, Bin Ding, Ziqiong Liu, Pratik Prabhanjan Brahma, Dong Li, Zicheng Liu, and Emad Barsoum. Geak: Introducing triton kernel ai agent & evaluation benchmarks, 2025. URL https://arxiv.org/abs/2507.23194.

Anjiang Wei, Tianran Sun, Yogesh Seenichamy, Hang Song, Anne Ouyang, Azalia Mirhoseini, Ke Wang, and Alex Aiken. Astra: A multi-agent system for gpu kernel performance optimization. In *NeurIPS 2025 Fourth Workshop on Deep Learning for Code*, 2025.

Zhongzhen Wen, Yinghui Zhang, Zhong Li, Zhongxin Liu, Linna Xie, and Tian Zhang. Multikernelbench: A multi-platform benchmark for kernel generation, 2025. URL https://arxiv.org/abs/2507.17773.

Shuxiao Xie, Shuyang Xie, Dezhi Ran, Wei Yang, and Tao Xie. AKO: Agentic kernel optimization. https://tongminglaic.github.io/AKO, 2026. Technical report.

Shanli Xing, Yiyan Zhai, Alexander Jiang, Yixin Dong, Yong Wu, Zihao Ye, Charlie Ruan, Yingyi Huang, Yineng Zhang, Liangsheng Yin, Aksara Bayyapu, Luis Ceze, and Tianqi Chen. Flashinfer-bench: Building the virtuous cycle for ai-driven llm systems, 2026. URL https://arxiv.org/abs/2601.00227.

Zijian Zhang, Rong Wang, Shiyang Li, Yuebo Luo, Mingyi Hong, and Caiwen Ding. Cudaforge: An agent framework with hardware feedback for cuda kernel optimization, 2025. URL https://arxiv.org/abs/2511.01884.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kober, Ying Sheng, et al. SGLang: Efficient execution of structured language model programs. 2024.

Jiace Zhu, Wentao Chen, Qi Fan, Zhixing Ren, Junying Wu, Xing Zhe Chai, Chotiwit Rungrueangwutthinon, Yehan Ma, and An Zou. Cudabench: Benchmarking llms for text-to-cuda generation, 2026. URL https://arxiv.org/abs/2603.02236.

Xinguo Zhu, Shaohui Peng, Jiaming Guo, Yunji Chen, Qi Guo, Yuanbo Wen, Hang Qin, Ruizhi Chen, Qirui Zhou, Ke Gao, et al. Qimeng-kernel: Macro-thinking micro-coding paradigm for llm-based high-performance gpu kernel generation. *arXiv preprint arXiv:2511.20100*, 2025.

Dongyun Zou, Zhekai Zhang, Shang Yang, and Hao Kang. KDA: Kernel design agents. https://github.com/mit-han-lab/kernel-design-agents, 2026. MLSys 2026 FlashInfer AI Kernel Generation Contest, full-agent track.

A BENCHMARK COMPOSITION

Table 3 reports the per-architecture composition of FASTKERNELS: the reference HuggingFace checkpoint, default inference dtype, and the number of L1 (primitive), L2 (fused/composite), L3 (layer/block) and L4 (end-to-end) tasks each architecture uses.

Counting rule. A task is one module file under `tasks/baseline/L`n`/` that defines at least one benchmarkable `nn.Module` and is reachable, through the import graph, from at least one of the 47 L4 architectures; it is counted once even when it ships a CUDA sidecar (17 tasks do) or defines several module classes. Twelve function-only helper kernels (e.g., Triton attention-merge and causal-convolution kernels) are called from inside other tasks and are optimized through them, so they are not counted separately. This yields 108 L1, 155 L2, and 74 L3 tasks plus 47 L4 models, 384 in total. L4 counts one task per end-to-end architecture: the encoder sub-models that some pipelines import (the CLIP and T5 text encoders, the Qwen2.5-VL vision encoder) are part of those pipelines' L4 tasks, and the L1–L3 set is identical with or without DeepSeek-V3.2, whose modules GLM-5.2 builds on. Because tasks are shared, the per-architecture columns of Table 3 sum to more than these totals.

Table 3: Full architecture composition of FASTKERNELS. L1–L4 report the number of primitive, fused/composite, layer/block-level, and end-to-end tasks each architecture uses, under the counting rule above (L4 is always 1 by construction). *Reference* is the HuggingFace checkpoint or upstream repository the L4 pipeline tracks; *dtype* is the default inference dtype.

Category	Architecture	Reference	dtype	L1	L2	L3	L4
Dense & MoE LLMs	Llama-3.1	meta-llama/Llama-3.1-8B-Instruct	BF16	13	6	1	1
	GLM-5.2	nvidia/GLM-5.2-NVFP4	NVFP4	31	11	1	1
	Mixtral	mistralai/Mixtral-8x7B-Instruct-v0.1	BF16	19	8	1	1
	BitNet 1.58b	microsoft/bitnet-b1.58-2B-4T	W1.58/BF16	13	5	1	1
	GPT-OSS (MXFP4)	openai/gpt-oss-120b	MXFP4	13	8	1	1
	EAGLE-3	jamesliu1/sglang-EAGLE3-Llama-3.1-Instruct-8B	BF16	13	4	0	1
	Gemma-4	google/gemma-4-26B-A4B-it	BF16	20	8	1	1
Linear Attention & New Archs	Mamba	state-spaces/mamba-2.8b-hf	BF16	7	3	1	1
	Mamba2	mistralai/Mamba-Codestral-7B-v0.1	BF16	9	3	1	1
	RWKV-7	fla-hub/rwkv7-2.9B-g1	BF16	11	2	1	1
	GLA	fla-hub/gla-2.7B-100B	BF16	11	2	1	1
	RetNet	fla-hub/retnet-2.7B-100B	BF16	11	2	1	1
	Qwen-3-Next	Qwen/Qwen3-Next-80B-A3B-Instruct	BF16	22	7	1	1
	Kimi-Linear	moonshotai/Kimi-Linear-48B-A3B-Instruct	BF16	25	9	1	1
	TTT-E2E	test-time-training/e2e	BF16	8	3	1	1
	Jamba	ai21labs/AI21-Jamba-Mini-1.7	BF16	20	8	1	1
Vision / Video / Audio	FLUX.1-Dev	black-forest-labs/FLUX.1-dev	BF16	16	10	3	1
	HunyuanVideo-1.5	hunyuanvideo-community/HunyuanVideo-1.5	BF16	24	15	4	1
	SDXL	stabilityai/stable-diffusion-xl-base-1.0	FP16	8	8	3	1
	SAM3.1	facebook/sam3.1	BF16	8	6	8	1
	Whisper	openai/whisper-large-v3	FP16	13	6	2	1
	CosyVoice3	FunAudioLLM/Fun-CosyVoice3-0.5B-2512	BF16	14	9	2	1
Multimodal & Encoders	Qwen2-VL	Qwen/Qwen2-VL-7B-Instruct	BF16	20	10	2	1
	Qwen3-VL-8B	Qwen/Qwen3-VL-8B-Instruct	BF16	27	13	3	1
	Qwen3-VL-235B	Qwen/Qwen3-VL-235B-A22B-Instruct-FP8	FP8	27	13	3	1
	Qwen-2.5-Omni	Qwen/Qwen2.5-Omni-7B	BF16	20	10	2	1
	SigLIP-2	google/siglip2-so400m-patch16-naflex	BF16	3	3	1	1
	DINOv3	facebook/dinov3-vit7b16-pretrain-lvd1689m	BF16	4	2	1	1
	SwinV2	timm/swinv2_large_window12_192.ms_in22k	FP32	4	3	2	1
Edge & Detection	MobileNetV4	timm/mobilenetv4_conv_medium.e500_r256_in1k	FP32	5	2	1	1
	ConvNeXtV2	facebook/convnextv2-base-22k-384	FP32	7	1	1	1
	EfficientNetV2	timm/efficientnetv2_rw_m.agc_in1k	FP32	6	3	1	1
	YOLOv10	jameslahm/yolov10n	FP16	8	11	3	1
	RTDetrV2	PekingU/rtdetr_v2_r101vd	FP32	16	8	4	1
3D / Robotics / Science	3DGS	nerfstudioteam/datasets (mip-NeRF 360)	FP32	3	1	0	1
	InstantNGP	nerfstudioteam/datasets (fox)	FP16	0	0	1	1
	PointTransformerV3	Pointcept/PointTransformerV3	FP32	3	1	0	1
	OpenFold3	OpenFold/OpenFold3	BF16	7	14	5	1
	Pi0	openpi/pi0_aloha_pen_uncap	BF16	13	5	2	1
	DP3	3d-diffusion-policy.github.io	FP32	8	3	1	1
Recommendation & Specialized	DLRMv2	criteo (1 TB click logs)	FP32	3	3	0	1
	LightGCN	MovieLens / Gowalla / Amazon-Book	FP32	2	1	0	1
	BGE-M3	BAAI/bge-m3	FP16	8	3	3	1
	ColBERTv2	colbert-ir/colbertv2.0	FP16	7	4	3	1
	LLaDA	GSAI-ML/LLaDA-8B-Instruct	BF16	6	3	1	1
World Models	Oasis	Etched/oasis-500m	BF16	7	7	5	1
	V-JEPA 2	facebook/vjepa2-vitl-fpc64-256	BF16	4	3	3	1

Note: A value of 0 indicates that the current implementation does not expose a benchmark task at that level. Quantized configurations reuse the same high-level task graph as their underlying architecture while activating distinct quantized kernels and execution paths.

B Interface Compatibility Examples

To illustrate the interface-compatible design, we show side-by-side comparisons of FastKernels module interfaces with their corresponding vLLM modules.

Example: Attention module.

FastKernels interface:

```
class Attention(nn.Module):
  def __init__(self, config,
               cache_config,
               quant_config):
    ...

  def forward(self,
              positions,
              hidden_states,
              kv_cache,
              attn_metadata):
    ...
```

vLLM interface:

```
class LlamaAttention(nn.Module):
  def __init__(self, config,
               cache_config,
               quant_config):
    ...

  def forward(self,
              positions,
              hidden_states,
              kv_cache,
              attn_metadata):
    ...
```

The matching constructor signatures and `forward` arguments enable direct module substitution. The same pattern extends to MLP blocks, normalization layers, and MoE routing modules.

Example: MoE layer.

FastKernels interface:

```
class MoELayer(nn.Module):
  def __init__(self, config,
               quant_config):
    ...

  def forward(self,
              hidden_states,
              router_logits):
    ...
```

vLLM interface:

```
class DeepseekV3MoE(nn.Module):
  def __init__(self, config,
               quant_config):
    ...

  def forward(self,
              hidden_states,
              router_logits):
    ...
```

C Scenario Sets

Scenario sets are YAML tables shipped with FastKernels. Each row fixes a checkpoint, tensor-parallel degree, dtype, and the workloads to run; per-workload datasets, request counts, and sequence lengths are defined once in a workload registry, so every tier and every run of a set sees identical inputs. Four sets ship with the benchmark:

- **full**: one scenario per architecture (47 scenarios, 184 workloads, TP 1–8, BF16/FP16/FP32/FP8/MXFP4/NVFP4), and therefore every task. It is the set on which we validate the baselines against production references (Section 6.1, Appendix G).
- **default**: the 11 scenarios of Table 4 (48 workloads), on which all agent results are reported. It contains at least one model per category, which MacroEval's per-family averaging requires, spans TP 1, 2, and 4 and BF16, FP16, FP8, and MXFP4 execution, and pairs throughput workloads (batched request mixes) with latency workloads (single requests or small fixed batches) for every model. Its B200 capture exercises 133 L1–L3 tasks (47 L1, 64 L2, 22 L3), which are the agents' targets; `oasis_rollout`, a whole-model rollout driver rather than a kernel, is excluded from scoring, leaving 21 L3 targets.
- **minimal**: Llama-3.1-8B with two workloads, a smoke test for the capture-to-bench pipeline.

- **vllm_only**: the 16 architectures whose reference is vLLM, used to compare vLLM releases for drift tracking (Appendix H).

Agents are scored on the default set rather than the full set because of cost: the long-horizon agents of Section 6.2 spend 1,763–2,512 agent-hours each on the default set's 133 operators (Appendix I), and the full set's architectures use more than twice as many L1–L3 tasks (337 versus 159 under the counting rule of Appendix A, of which the default workloads invoke 133 on B200).

Table 4: **The default scenario set.** Workloads are listed by name; their datasets and sizes are defined in the workload registry (Appendix F).

Model	Category	TP	dtype	Workloads
Llama-3.1-8B-Instruct	Dense & MoE LLMs	1	BF16	mixed, long-context, single request, fixed batch 32
gpt-oss-120b	Dense & MoE LLMs	2	MXFP4	mixed, long-context, single request, fixed batch 32
Qwen3-Next-80B-A3B-Instruct	Linear Attention	2	BF16	mixed, long-context, single request, fixed batch 32
Kimi-Linear-48B-A3B-Instruct	Linear Attention	2	BF16	mixed, long-context, single request, fixed batch 32
GLA-2.7B	Linear Attention	1	BF16	mixed, long-context, single request, fixed batch 32
FLUX.1-dev	Vision / Video / Audio	1	BF16	512 and 1024 px, batched and single image
Qwen3-VL-235B-A22B-Instruct-FP8	Multimodal	4	FP8	text only, image, video, single image, single video
YOLOv10n	Edge & Detection	1	FP16	COCO val, single image, batch 4
OpenFold3	3D / Robotics / Science	1	BF16	four length buckets, batched and single
BGE-M3	Recommendation	1	FP16	MLDR documents, single request, fixed batch 32
Oasis-500M	World Models	1	FP16	four rollout / denoising settings, batch-1 latency

D Workload Capture and Case Selection

Instrumentation. `fastkernels capture <set>` loads each scenario into its own engine process (a TP-N scenario claims N GPUs; scenarios are packed onto the GPU pool and isolated so that a crash in one does not affect the others) and runs every listed workload on its real inputs. Before the model is built, it wraps the constructor and `forward` of every `nn.Module` class under `tasks/baseline`. A constructor call is recorded as a reconstructable *recipe*: configuration dataclasses and enums with their field values, dtypes, tensor shapes, and references to other captured operators (e.g., the linear layers inside an attention block); runtime handles such as process groups are marked opaque. A forward call is recorded as the shape, dtype, and, for non-contiguous tensors, stride of every tensor argument, together with the values of scalar arguments. Shape capture stores no weights or activation values; golden inputs for data-dependent operators are recorded separately (below).

Deduplication. Each call is keyed by the canonical serialization of its record, and identical keys are merged into one *variant* that keeps a call count. Each forward variant also records which constructor variants built the instances it ran on, so the benchmark can rebuild exactly the module configuration that produced each shape. Since every layer of a model and every decode step with the same batch composition produce the same key, the 12.4M L1–L3 forward calls of the default B200 capture reduce to 93.3K variants across its 48 per-workload reports, and to 87.7K distinct (constructor, forward) cases once merged across reports. The distribution is long-tailed: half of the operators have at most seven distinct cases, while operators whose shapes follow the per-step token count (decode attention, RMSNorm, linear layers, embeddings) reach thousands. For the few operators that read the global inference context inside `forward` (MLA attention), a coarse context summary (prefill or decode, and a power-of-two bucket of the number of prefill sequences) is folded into the key, which is enough to rebuild the code path without defeating deduplication.

Verification. Two independent checks run after every workload. First, a forward pre-hook, registered separately from the wrappers, re-observes every operator call; its per-operator keys and counts must match the wrapper records. Second, a mock continuous-batching and chunked-prefill scheduler, driven only by each request's real prompt and generated lengths, the engine's token and sequence budgets, and its KV-block pool, replays the serving schedule and predicts the per-step batch composition; the captured leading dimensions (token counts, sequence counts, batch sizes) must agree with it. The replay does not model speculative decoding, so only the first check applies to EAGLE-3.

Case selection. `fastkernels bench` loads every report of a capture and deduplicates cases across reports by (constructor recipe, forward record). When an operator has more than five distinct cases (the default budget), it keeps the smallest, median, and largest by total input elements and fills the remaining slots with the most frequently called cases, so the selection covers the size range rather than only the hot decode shape. On the default B200 capture this yields 582 benchmarked cases over the 154 operator classes defined in the 133 target tasks.

Input materialization. Floating-point inputs are drawn from a standard normal (FP8 inputs are clamped normals cast to FP8) and masks are all ones. Integer tensors are made valid for the call: cumulative sequence-length vectors are rebuilt as partitions consistent with the query and key token counts, token and position indices are bounded by the vocabulary size and maximum position, expert indices by the expert count, and slot mappings and block tables by the cache they index. Captured non-contiguous layouts are reproduced with the recorded strides. Baseline and candidate share randomly initialized weights; quantized weights (block-scaled FP8, NVFP4, MXFP4) are produced with the model's own quantization and weight-preparation routines, so kernels see the layouts they would see after checkpoint loading. For data-dependent operators such as MoE dispatch, whose control flow depends on input values (Appendix E), the kernel tier can instead replay golden inputs captured from real end-to-end executions, so that correctness and performance are measured on the tensors the model actually produces. Cases whose inputs the baseline rejects, or whose baseline output is all zeros (so that a stub would pass), are skipped. The all-reduce and row-parallel linear run in an NCCL group spawned with world size equal to the capture's TP degree, with every rank executing the same seeded cases.

Correctness and timing. Each case runs three rounds of fresh inputs, and an output passes when at least 99% of its elements are within per-dtype tolerances ($\mathrm{atol}, \mathrm{rtol}$ of $10^{-5}, 10^{-3}$ for FP32; $10^{-2}, 10^{-2}$ for FP16 and BF16; $0.125, 0.125$ for FP8). Timing uses CUDA events with 10 warm-up and 50 timed iterations, flushing the L2 cache and rotating input buffers between iterations, and reports the median. Each operator runs in its own process, where the harness checks the identity of the timing primitive and of its own critical functions, background-thread counts, and output types, and reports any tampering as a reward hack.

E Case Study: Captured Inputs Matter for Data-Dependent Ops

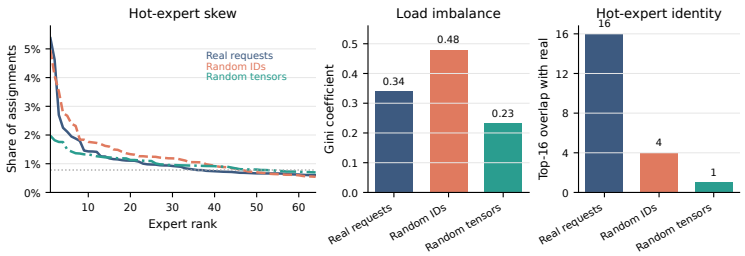


Figure 3: **Input capture changes data-dependent execution.** First MoE layer of Qwen3-VL-30B-A3B-Instruct-FP8 (128 experts, top-8 routing) under three inputs: real WildChat requests, random token IDs (matched lengths), and random tensors injected at the gate. Random tensors look near-uniform; random tokens skew differently from real requests, with only 4 of the top-16 hot experts in common.

Many production operators have data-dependent control flow—which experts, sparse blocks, cache slots, or communication paths execute is determined by the input. We isolate this effect on a well-known case: the first MoE gate of Qwen3-VL-30B-A3B-Instruct-FP8 (Figure 3). Random tensors yield a near-uniform gate (Gini 0.232); random token IDs over-concentrate load and identify different hot experts from real WildChat requests (Gini 0.480 vs. 0.339; only $4/16$ shared hot experts). FastKernels's capture-and-replay path therefore rewards data-dependent kernels on the tensors production requests actually create, rather than on synthetic control flow: the kernel tier can replay

golden inputs captured from real end-to-end executions (Appendix D), and the end-to-end tier runs real requests.

F EXPERIMENTAL DETAILS

Hardware. End-to-end L4 validate results in Section 6.1 and Tables 5–6 are measured on a NVIDIA DGX B200 box with 8x NVIDIA B200 SXM GPUs, 180 GB HBM3e each (1.44 TB total), interconnected via 5th-gen NVLink/NVSwitch. Single-GPU experiments use one GPU; multi-GPU experiments use up to $8\times$B200 (tensor-parallel degree matching the production reference: e.g. TP=2 Mixtral/GPT-OSS, TP=8 GLM-5.2). L1–L3 agent captures in Section 6.2 use the same B200 nodes. The Hopper results in Appendix G.2 repeat the same L4 scenario table on a node with 8x NVIDIA H200 SXM GPUs, 141 GB HBM3e each, using the same tensor-parallel degrees; GLM-5.2 is not run there because its NVFP4 kernels require Blackwell.

Serving benchmark. Autoregressive L4 throughput uses a mixed-length request mix and a long-context mix against the production server (vLLM, SGLang, or FLA); latency uses single-request (batch 1) and fixed-batch-32 decoding. Each latency measurement is the median of 5 iterations. Non-AR models use family-specific public workloads (Parti-prompts, SEED-TTS-Eval, LibriSpeech, COCO val, Food101, MS MARCO, ALOHA, etc.) as listed in Tables 5 and 6.

G REFERENCE PERFORMANCE AND CORRECTNESS DETAILS

Table 5 and Table 6 report the B200 L4 validate run for every architecture; Appendix G.2 repeats it on H200. For each row we list the production reference, the public workload mix, the geometric mean of throughput and latency speedups over that mix, and the alignment metric against the reference. Autoregressive rows summarize mean prefix-match length and exact-match rate across throughput scenarios; cosine, top-1, box/label, NLL, and pass flags are reported as published by the corresponding harness. If a model has no latency workload, Lat. is "—". Per-row geometric means match the aggregator used in MACROEVAL (Section 5) and in Section 6.1.

G.1 BLACKWELL (B200) RESULTS

Table 5: End-to-end B200 validate results for FASTKERNELS architectures (Part I). *Thr.* and *Lat.* are geometric-mean speedups versus the named production reference. *Correctness* is the harness alignment summary (prefix-match length and exact-match rate for autoregressive models; cosine, top-1, or pass flags otherwise).

Category	Architecture	Reference	Workload	Thr.	Lat.	Correctness
Dense & MoE LLMs	Llama-3.1	`vLLM`	WildChat mixed / long-context	0.95×	1.02×	prefix 136.3 / EM 28.6%
	Mixtral	`vLLM`	WildChat mixed / long-context	1.06×	1.00×	prefix 86.3 / EM 12.2%
	GLM-5.2 (NVFP4)	`vLLM`	WildChat mixed / long-context	1.41×	0.94×	prefix 33.5 / EM 8.9%
	BitNet 1.58b	Microsoft BitNet GPU	WildChat mixed / long-context	1.10×	1.20×	prefix 122.1 / EM 11.0%
	GPT-OSS (MXFP4)	`vLLM`	WildChat mixed / long-context	0.95×	0.95×	prefix 53.8 / EM 8.2%
	EAGLE-3	`SGLang`	WildChat EAGLE-3 speculative decoding	0.93×	1.04×	prefix 160.6 / EM 37.5%
	Gemma-4	`vLLM`	WildChat mixed / long-context	1.00×	1.09×	prefix 70.0 / EM 9.6%
Linear Attention & New Archs	Mamba	`vLLM`	WildChat mixed / long-context	2.10×	2.02×	prefix 129.5 / EM 31.9%
	Mamba2	`vLLM`	WildChat mixed / long-context	0.96×	0.95×	prefix 203.1 / EM 54.6%
	RWKV-7	`FLA`	mixed / long-context serving	1.22×	1.26×	prefix 100.7 / EM 14.8%
	GLA	`FLA`	mixed / long-context serving	1.44×	1.75×	prefix 262.1 / EM 70.9%
	RetNet	`FLA`	mixed / long-context serving	3.40×	1.73×	prefix 94.4 / EM 29.9%
	Qwen-3-Next	`vLLM`	mixed / long-context serving	1.21×	0.99×	prefix 11.7 / EM 1.4%
	Kimi-Linear	`vLLM`	mixed / long-context serving	1.21×	1.04×	prefix 63.8 / EM 9.2%
	TTT-E2E	JAX ref.	pretrain + meta forward	1.95×	1.95×	token NLL Δ 0.022
	Jamba	`vLLM`	mixed / long-context serving	0.84×	1.10×	prefix 91.4 / EM 15.6%
Vision / Video / Audio	FLUX.1-Dev	`vllm-omni`	Parti-prompts; 512 / 1024	1.31×	1.10×	min cos 0.949
	HunyuanVideo-1.5	`vllm-omni`	480p short / medium T2V	1.01×	1.11×	min cos 0.959
	SDXL	`diffusers`	Parti-prompts; 512 / 1024	1.03×	1.02×	min cos 0.919
	SAM3.1	`sam3`	image pipeline + video frames	1.04×	1.06×	pass / min cos 0.871
	Whisper	`vLLM`	LibriSpeech test.clean	1.07×	1.03×	prefix 385.9 / EM 63.5%
	CosyVoice3	`vllm-omni`	SEED-TTS-Eval; short / medium / long	2.30×	2.34×	mel cos 0.991
Multimodal & Encoders	Qwen2-VL	`vLLM`	text / image / video	1.24×	1.01×	prefix 246.0 / EM 43.9%
	Qwen3-VL-8B	`vLLM`	text / image / video	1.26×	1.05×	prefix 141.1 / EM 20.2%
	Qwen3-VL-235B (FP8)	`vLLM`	text / image / video	1.59×	0.95×	prefix 40.1 / EM 2.9%
	Qwen-2.5-Omni	`vLLM`	text / image / video / audio	1.38×	1.07×	prefix 161.4 / EM 26.9%
	SigLIP-2	`timm`	default-res + high-res	1.01×	1.06×	min cos 1.000
	DINOv3	`timm`	default-res + high-res	0.94×	0.94×	min cos 1.000
	SwinV2	`timm`	default-res + high-res	0.94×	0.91×	min cos 1.000

Table 6: End-to-end B200 validate results for FASTKERNELS architectures (Part II).

Category	Architecture	Reference	Workload	Thr.	Lat.	Correctness
Edge & Detection	MobileNetV4	`timm`	default-res + high-res	1.03×	1.04×	min cos 1.000
	ConvNeXtV2	`transformers`	Food101 validation	1.00×	1.04×	top-1 1.000
	EfficientNetV2	`timm`	Food101 validation	1.04×	1.09×	top-1 1.000
	YOLOv10	THU-MIG/ yolov10	COCO val detection	1.14×	0.94×	box 0.997 / score 1.000 / lbl 100%
	RTDetrV2	`transformers`	COCO val detection	0.97×	0.90×	box 0.928 / score 1.000 / lbl 73.4%
3D / Robotics / Science	3DGS	`gsplat`	deterministic orbit render	1.00×	0.99×	pass / RGB cos 1.000
	InstantNGP	`pyngp`	fox render	0.98×	0.98×	pass / RGBA cos 1.000
	PointTransformerV3	PTv3 ref.	ScanObjectNN + batch-8	0.98×	0.99×	feat. cos 0.986 (fail)
	OpenFold3	OpenFold3 ref.	4 length buckets	1.01×	1.02×	align 100%
	Pi0	`OpenPI`	ALOHA 3-cam / 1-cam	3.09×	4.17×	pass / action cos 1.000
	DP3	3D-Diffusion-Policy	1-env + batch-8	1.13×	1.13×	min cos 1.000
Recommendation & Specialized	DLRMv2	`TorchRec`	Adult Census CTR batch	1.10×	1.37×	cosine undefined
	LightGCN	`PyG`	recommend-batch	1.02×	1.04×	pass / cos 1.000
	BGE-M3	`vLLM`	MLDR documents	2.35×	0.97×	pass / cos 0.999
	ColBERTv2	`vLLM`	MS MARCO passages	3.13×	1.33×	pass / cos 1.000
	LLaDA	`Fast-dLLM`	HumanEval Fast-dLLM dual	0.96×	0.96×	prefix 168.6 / EM 56.1%
World Models	Oasis	`open-oasis`	Minecraft clips; 4 DDIM settings	1.17×	1.16×	pass / min cos 0.997
	V-JEPA 2	`transformers`	predictor / encoder	0.97×	0.98×	pass / cos 1.000

Note: Speedup values are geometric means over all validate scenarios for that row; values below 1 indicate slower execution than the reference. CosyVoice3 e2e correctness is the overall median mel-spectrogram cosine from the run log (the compact markdown table left this field blank). DLRMv2 reports `pass=False` with NaN cosine against TorchRec on this checkpoint; PointTransformerV3 reports `pass=False` at feature cosine 0.986. RetNet's 3.40× throughput geomean still mixes a 6.83× mixed scenario (prefix 21.2) with 1.69× long-context (prefix 167.6).

G.2 HOPPER (H200) RESULTS

Tables 7 and 8 repeat the L4 validate run on 8×H200 (Appendix F) with the same scenario table, workloads, references, and aggregation as the B200 tables above. Figure 4 is the H200 counterpart of Figure 2, and Figures 5 and 6 show the per-architecture speedups. Over the 46 architectures that run on both GPUs, the throughput geomean is 1.19× on H200 versus 1.22× on B200, and the latency geomean is 1.15× on both. The four largest H200 throughput gains (ColBERTv2, RetNet, Pi0, CosyVoice3) are also among the five largest on B200, and dense and MoE LLM serving stays at parity (0.99×); the largest drops relative to B200 are Mamba (2.10× → 1.08×) and BGE-M3 (2.35× → 1.37×).

Table 7: End-to-end H200 validate results for FASTKERNELS architectures (Part I). Columns as in Table 5. †Re-run serially on a second 8×H200 node (see the note under Table 8).

Category	Architecture	Reference	Workload	Thr.	Lat.	Correctness
Dense & MoE LLMs	Llama-3.1	vLLM	WildChat mixed / long-context	0.97×	1.00×	prefix 140.3 / EM 28.5%
	Mixtral	vLLM	WildChat mixed / long-context	1.02×	1.01×	prefix 72.9 / EM 9.7%
	GLM-5.2 (NVFP4)	vLLM	WildChat mixed / long-context	—	—	not run (NVFP4 is Blackwell-only)
	BitNet 1.58b	Microsoft BitNet GPU	WildChat mixed / long-context	1.01×	1.10×	prefix 108.2 / EM 7.8%
	GPT-OSS (MXFP4)	vLLM	WildChat mixed / long-context	1.04×	0.92×	prefix 63.0 / EM 8.8%
	EAGLE-3†	SGLang	WildChat EAGLE-3 speculative decoding	0.96×	1.04×	prefix 179.2 / EM 50.0%
	Gemma-4	vLLM	WildChat mixed / long-context	0.92×	1.03×	prefix 72.3 / EM 7.3%
Linear Attention & New Archs	Mamba	vLLM	WildChat mixed / long-context	1.08×	1.03×	prefix 147.7 / EM 36.9%
	Mamba2	vLLM	WildChat mixed / long-context	0.99×	0.96×	prefix 207.1 / EM 53.5%
	RWKV-7	FLA	mixed / long-context serving	1.34×	1.29×	prefix 99.8 / EM 12.7%
	GLA	FLA	mixed / long-context serving	1.34×	1.67×	prefix 264.1 / EM 71.1%
	RetNet	FLA	mixed / long-context serving	3.05×	1.75×	prefix 98.8 / EM 32.2%
	Qwen-3-Next	vLLM	mixed / long-context serving	1.02×	0.99×	prefix 11.4 / EM 1.3%
	Kimi-Linear	vLLM	mixed / long-context serving	1.07×	0.89×	prefix 18.3 / EM 4.0%
	TTT-E2E	JAX ref.	pretrain + meta forward	1.75×	1.75×	token NLL Δ 0.022
	Jamba†	vLLM	mixed / long-context serving	0.96×	1.06×	prefix 98.3 / EM 16.4%
Vision / Video / Audio	FLUX.1-Dev	vllm-omni	Parti-prompts; 512 / 1024	1.11×	1.02×	min cos 0.981
	HunyuanVideo-1.5	vllm-omni	480p short / medium T2V	1.00×	1.10×	min cos 0.943
	SDXL	diffusers	Parti-prompts; 512 / 1024	1.01×	1.02×	min cos 0.923
	SAM3.1	sam3	image pipeline + video frames	1.13×	1.10×	pass / min cos 0.871
	Whisper	vLLM	LibriSpeech test.clean	1.04×	0.95×	prefix 386.1 / EM 63.6%
	CosyVoice3	vllm-omni	SEED-TTS-Eval; short / medium / long	2.85×	2.05×	mel cos 0.991
Multimodal & Encoders	Qwen2-VL	vLLM	text / image / video	1.13×	0.99×	prefix 248.2 / EM 44.8%
	Qwen3-VL-8B	vLLM	text / image / video	1.21×	1.02×	prefix 139.5 / EM 19.9%
	Qwen3-VL-235B (FP8)	vLLM	text / image / video	1.56×	0.98×	prefix 41.9 / EM 2.8%
	Qwen-2.5-Omni	vLLM	text / image / video / audio	1.43×	1.01×	prefix 158.9 / EM 25.0%
	SigLIP-2†	timm	default-res + high-res	1.00×	1.06×	min cos 0.986
	DINOv3†	timm	default-res + high-res	0.94×	0.89×	min cos 1.000
	SwinV2†	timm	default-res + high-res	1.00×	1.04×	min cos 1.000

Table 8: End-to-end H200 validate results for FASTKERNELS architectures (Part II).

Category	Architecture	Reference	Workload	Thr.	Lat.	Correctness
Edge & Detection	MobileNetV4†	`timm`	default-res + high-res	1.04×	1.11×	min cos 1.000
	ConvNeXtV2	`transformers`	Food101 validation	1.00×	0.99×	top-1 1.000
	EfficientNetV2	`timm`	Food101 validation	1.06×	0.99×	top-1 1.000
	YOLOv10	THU-MIG/ yolov10	COCO val detection	1.04×	2.61×	box 0.461 / score 0.549 / lbl 94.6% (fail)
	RTDetrV2	`transformers`	COCO val detection	1.00×	0.85×	box 0.928 / score 1.000 / lbl 73.4%
3D / Robotics / Science	3DGS	`gsplat`	deterministic orbit render	1.00×	1.00×	pass / RGB cos 1.000
	InstantNGP	`pyngp`	fox render	0.99×	0.99×	pass / RGBA cos 1.000
	PointTransformerV3	PTv3 ref.	ScanObjectNN + batch-8	1.00×	0.99×	feat. cos 0.986 (fail)
	OpenFold3	OpenFold3 ref.	4 length buckets	1.00×	0.95×	align 100%
	Pi0	`OpenPI`	ALOHA 3-cam / 1-cam	2.98×	3.71×	pass / action cos 1.000
	DP3	3D-Diffusion-Policy	1-env + batch-8	1.14×	1.17×	min cos 1.000
Recommendation & Specialized	DLRMv2	`TorchRec`	Adult Census CTR batch	1.07×	1.34×	cosine undefined
	LightGCN	`PyG`	recommend-batch	1.02×	1.04×	pass / cos 1.000
	BGE-M3	`vLLM`	MLDR documents	1.37×	1.04×	pass / cos 0.997
	ColBERTv2	`vLLM`	MS MARCO passages	3.77×	2.59×	pass / cos 1.000
	LLaDA	`Fast-dLLM`	HumanEval Fast-dLLM dual	1.04×	1.05×	prefix 166.2 / EM 53.7%
World Models	Oasis	`open-oasis`	Minecraft clips; 4 DDIM settings	1.17×	1.13×	pass / min cos 0.996
	V-JEPA 2	`transformers`	predictor / encoder	0.99×	1.00×	pass / cos 1.000

Note: Aggregation as in Tables 5–6. †EAGLE-3, Jamba, and the four `timm` encoders failed in the parallel run for environment reasons (an SGLang 0.5.20 `ServerArgs` API change, a CUDA illegal-memory-access crash that did not reproduce, and gated ImageNet access, respectively) and were re-run serially on a second 8×H200 node with the same software stack; the EAGLE-3 re-run also needed a two-line compatibility fix in the SGLang harness. YOLOv10 fails alignment on H200 (box cosine 0.461 vs. 0.997 on B200), so its H200 speedups, notably the 2.61× latency, are not like-for-like and should not be read as gains. DLRMv2 again reports a degenerate cosine (0 here, NaN on B200) with a maximum absolute difference of 0; PointTransformerV3 fails at feature cosine 0.986 as on B200. RetNet's 3.05× throughput geomean mixes a 5.61× mixed scenario (prefix 19.1) with 1.66× long-context (prefix 178.4).

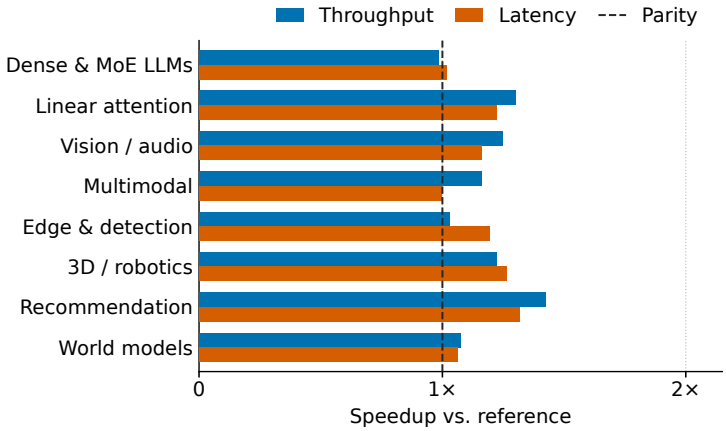


Figure 4: **End-to-end speedup by category on H200.** Counterpart of Figure 2: each bar is the geometric mean over the architectures of that category in Tables 7 and 8 (GLM-5.2 is not run). The edge & detection latency bar includes YOLOv10, which fails alignment on H200; without it, that bar is 0.98×.

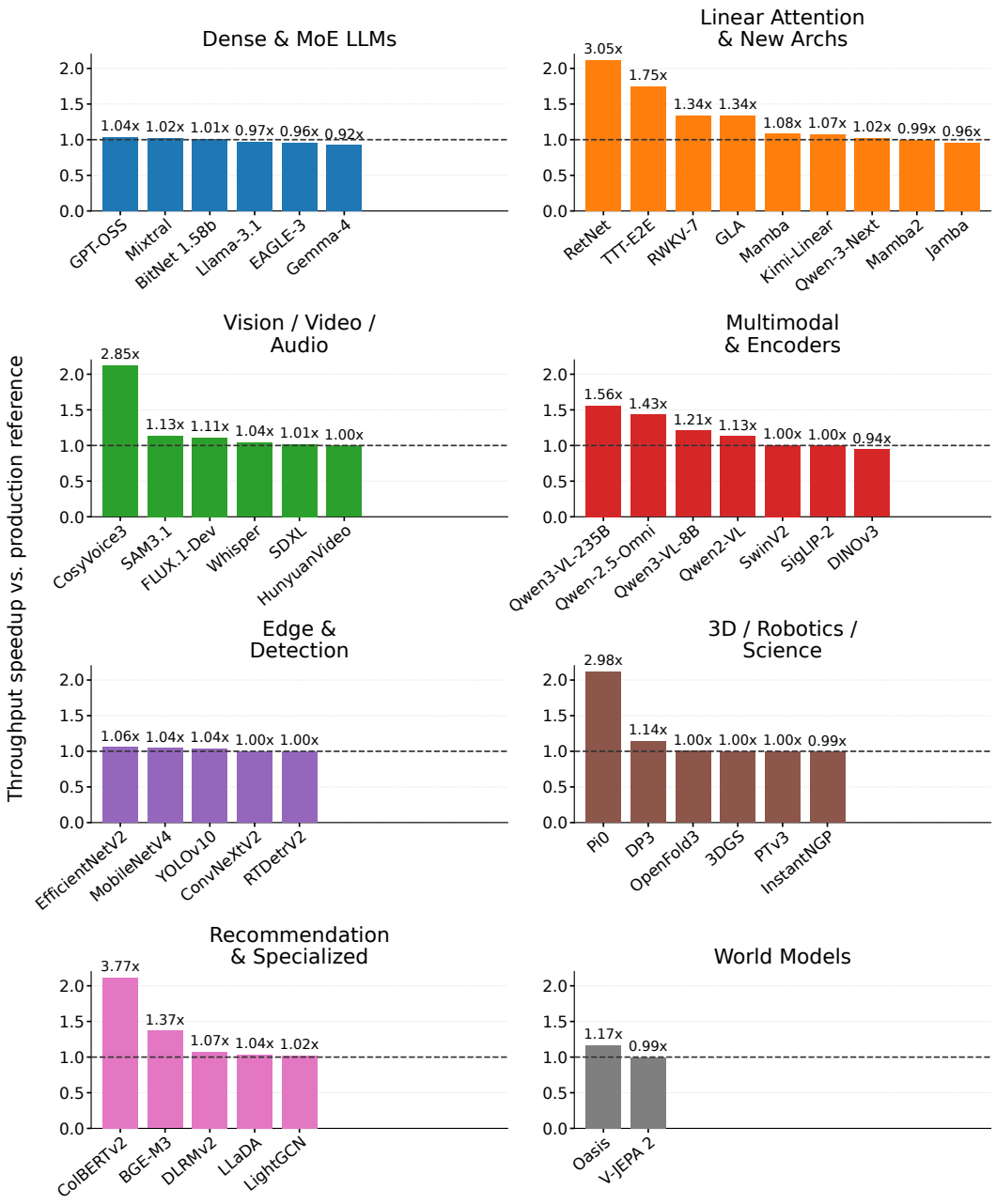


Figure 5: **Per-architecture throughput speedup on H200**, grouped by category. Bars taller than the axis are clipped and labeled with their value; the dashed line marks parity.

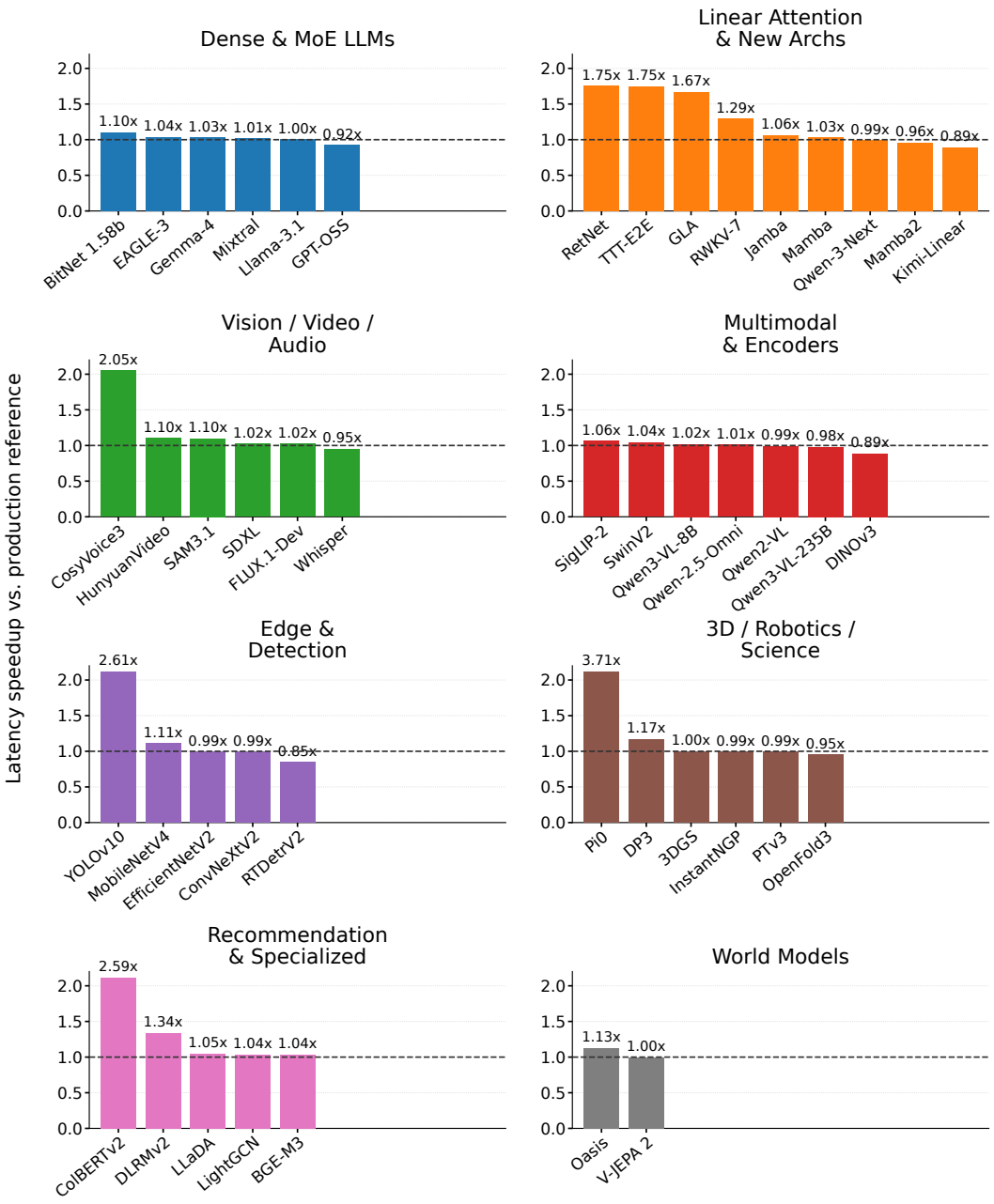


Figure 6: **Per-architecture latency speedup on H200**, drawn as in Figure 5. YOLOv10's 2.61× comes from a run that fails alignment (Table 8).

H Baseline Versioning and Drift

Production kernels are a moving target, so FastKernels is versioned: each release pins the exact revision of every reference library (vLLM, FlashInfer, FLA, diffusers, . . .) per target GPU, and the results in this paper use vLLM 0.26.0 (released 2026-07-25). A speedup over a FastKernels baseline is therefore a speedup over production at that release. To track drift, FastKernels ships a daily CI job (`scripts/baseline_drift.sh`): whenever a new vLLM release appears, it re-benchmarks only the reference models under that release on each pinned GPU type, in isolated environments with one job per node, and computes per-family drift coefficients (per-model geometric mean over workloads, models weighted equally within a family). Dividing a speedup measured

against the pinned baseline by the family coefficient estimates the speedup against the newer release; when any family moves by more than 10%, a re-release with re-pinned references is prepared.

To estimate the cadence this implies, we ran the same measurement over the 127 days between vLLM 0.18.0 (2026-03-20) and 0.26.0 on 8×H200 and 8×B200, on the 13 reference models both releases serve, with identical workloads (Table 9). The median model moved by only $1.03\times$ (throughput) and $1.01\times$ (latency) on H200, and by $1.07\times$ and $1.09\times$ on B200. Drift is concentrated rather than uniform: one family crossed the 10% threshold on H200 (linear attention and SSMs, $1.15\times$ throughput) and two on B200, where the LLM family is dominated by gpt-oss-120b's MXFP4 path maturing on Blackwell ($2.8\times$, against $1.04\times$ on H200). Newer releases are not uniformly faster: Qwen3-VL-235B-FP8 lost throughput on both GPUs ($0.83\times$ and $0.87\times$). This implies a re-release at least every four months, sooner on newer hardware whose support for new number formats is still maturing—comparable to the cadence at which any benchmark that stays current must add newly released architectures.

Table 9: **Reference drift from vLLM 0.18.0 to 0.26.0** (127 days), as computed by the baseline-drift CI job: speedup of the newer release (>1: faster), per-model geometric mean over workloads, models weighted equally within a family. Latencies are 5-iteration medians ($<1\%$ spread); throughput is one timed pass per workload, which three repeated runs per release of Qwen3-VL-235B-FP8 place within 1.5%. On B200, six models needed `nvidia-cutlass-dsl==4.5.3` to import under vLLM 0.18.0.

		H200		**B200**	
Model family	n	Thr.	Lat.	Thr.	Lat.
LLM (dense + MoE)	3	1.09×	1.08×	1.44×	1.37×
Linear attention / SSM / hybrid	5	1.15×	1.03×	1.14×	1.18×
Multimodal / VLM	4	0.98×	1.05×	1.01×	1.07×
Audio	1	1.02×	1.02×	1.02×	1.07×
Median model	13	1.03×	1.01×	1.07×	1.09×

I Agent Selection, Configuration, and Cost

Selection. We chose agents to cover the three lines of work that currently produce the strongest kernels. From the KernelBench ecosystem we run Dr. Kernel (Liu et al., 2026), one of its most recent agents; of the others we tried, KernelAgent (Cheng et al., 2026) did not reproduce its reported results in our setting, and the rest assume KernelBench's single-file tasks with a self-contained reference and cannot load production-shaped modules that require configuration objects, KV-cache wrappers, packed quantized weights, or a distributed context. From the MLSys 2026 FlashInfer AI Kernel Generation Contest (FlashInfer Team, 2026), whose tasks were drawn from production LLM workloads and scored against FlashInfer's expert kernels, we run KDA (Zou et al., 2026), whose full-agent entry placed first on MoE, second on DSA, and third on GDN, and AKO (Xie et al., 2026) (its AKO4X harness was adapted by the GEMM People entry, which placed second on MoE), whose release reports beating the FlashInfer expert kernels on 9 of 10 operator families on B200. Finally, we run Claude Code and Codex (OpenAI, 2025) directly, with no harness. Because KDA and AKO drive the same Claude model as our Claude Code runs (KDA adds a GPT reviewer), comparing them with Claude Code isolates the contribution of the harness.

Configurations. Table 10 summarizes how each agent was run. All agents receive the same task: the baseline module, its captured cases, and `fastkernels bench` on a GPU that is leased only while they benchmark or profile. A candidate becomes a *winner* only if every non-skipped case passes and its geometric-mean speedup is at least $1\times$; an unchanged baseline that passes counts as correct at $1\times$ but not as a fast_1 win. The independent campaign optimizes all 133 operators in standalone mode; the sequential campaign regenerates the 86 L2 and L3 operators on top of the frozen lower-level winners.

Table 10: **Agent configurations.** *Cap*: hang cap on one agent session (the candidate is kept on timeout), not a target budget.

Agent	Model(s)	Loop per operator	Cap
Dr. Kernel	released `drkernel-14b`	8 samples × 3 generate–execute–feedback turns (temperature 1.0, top-p 0.95)	—
Claude Code	Claude Opus 5 (1M context, high effort)	one session with the default tools	5 h
Codex	GPT-5.6 sol (high)	one session, same prompt as Claude Code	5 h
KDA	Claude Opus 5 (draft, plan, implement); GPT-5.6 sol (review)	one planning cycle, then two implement–review rounds, with KernelWiki and Nsight Compute skills	6 h
AKO	Claude Opus 5 (master and sub-agents)	two master-directed rounds, each a fresh sub-agent rewriting the kernel along one direction; improving kernels are archived	5 h/round

Budget and the number of agents. Table 11 reports what each agent spent: 6,914 agent-hours and 41.9B tokens (266M output tokens) in total, of which Claude Code, KDA, and AKO account for 1,763, 2,512, and 2,234 agent-hours. These are the budgets at which such agents are used in practice: the MLSys 2026 contest gave teams 13 weeks, from the release of the starter kit (January 22) to the kernel deadline (April 24), to optimize five kernels (FlashInfer Team, 2026). A leaderboard that samples many models for a few turns per task is far cheaper per entry, but it measures a different system than the long-horizon agents practitioners deploy, and the ranking of those agents is what FASTKERNELS is meant to report. Each additional long-horizon agent costs about 2,000 agent-hours and 8–22B tokens on the default set. Codex is the exception worth noting: its kernel-level results are comparable to the Claude-based agents' at 321 agent-hours. FASTKERNELS imposes no structure on the agent, so any agent that writes candidate files can be scored with the same harness.

Evolutionary search. AlphaEvolve (Novikov et al., 2025) and KernelEvolve (Liao et al., 2026) are not publicly available. OpenEvolve (Sharma, 2025) is, and was the organizers' agent baseline in the MLSys 2026 contest, but evolutionary frameworks evaluate a population of candidates in every generation, and on FASTKERNELS each evaluation compiles a kernel and benchmarks it on its production-shaped cases (for collectives, on a full tensor-parallel group); their cost per operator therefore scales with population size times the number of generations, on top of the LLM calls that produce each candidate, which puts a 133-operator campaign beyond our budget. The AKO authors also report that, during the contest, search-based methods performed far worse than letting a coding agent optimize kernels directly (Xie et al., 2026).

Table 11: **Agent cost per campaign** (133 operators independently, 86 regenerated sequentially). *Agent-hours*: sum over operators of each operator's wall-clock generation time, including relaunches; operators run in parallel, so this is not campaign wall-clock time. *Tokens* as reported by each agent's CLI or API: input, output, and cached (re-read) context; output includes reasoning tokens. Dr. Kernel runs a local model without prompt caching; KDA counts both its Claude and Codex sessions.

			Agent-hours		Tokens		
Agent	Campaign	Ops	Total	Median/op	Total (B)	Output (M)	Cached (%)
Dr. Kernel	Independent	133	53	0.26	0.03	10.4	—
	Sequential	86	32	0.34	0.02	8.0	—
Claude Code	Independent	133	1,171	7.27	4.35	31.4	97
	Sequential	86	592	6.06	3.58	22.5	97
KDA	Independent	133	1,751	12.33	12.31	62.8	96
	Sequential	86	762	8.22	9.48	42.2	97
AKO	Independent	133	1,421	9.43	5.47	47.4	97
	Sequential	86	813	8.70	4.86	35.3	97
Codex	Independent	133	253	1.86	1.02	4.0	98
	Sequential	86	67	0.71	0.74	2.4	98

J Additional Kernel-Level Results

Figure 7 extends the fast_1 and fast_2 columns of Table 1 to every speedup threshold p, for the independent mode.

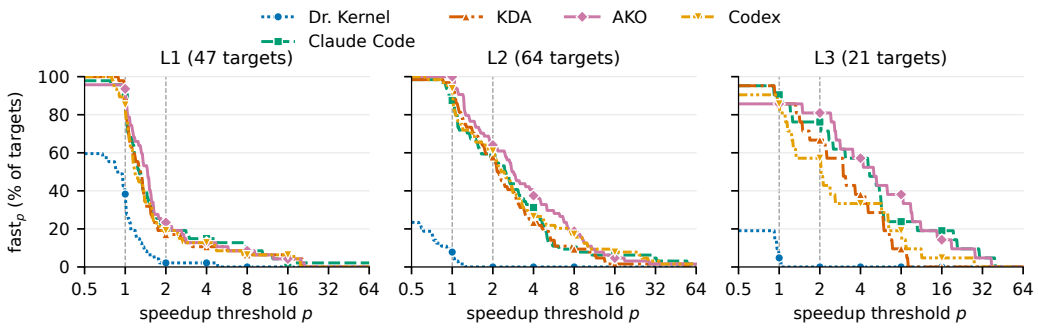


Figure 7: **fast$_p$ by level (independent mode).** Share of targets whose kernel is correct and more than $p\times$ faster than the production baseline; the value at the left edge approaches the share of correct kernels. Dashed lines mark $p = 1$ and $p = 2$.

Table 12 compares the same operators across the three kernel-bench modes (Section 6.2).

Table 12: **Composition effect and sequential generation**, over operators correct in both modes (n). *Ratio*: geometric mean of the per-operator speedup ratio (composed / independent, or sequential / composed). *Broke*: independent winners that fail correctness once composed.

	Composition effect (Independent → Composed)						**Sequential** (Composed → Seq.)			
	L2			L3			L2		L3	
Agent	n	Ratio	Broke	n	Ratio	Broke	n	Ratio	n	Ratio
Dr. Kernel	5	0.99×	0	1	1.00×	0	1	1.78×	0	—
Claude Code	54	1.09×	2	14	0.99×	4	50	0.99×	14	0.97×
KDA	51	0.98×	10	11	0.88×	6	47	0.90×	11	1.07×
AKO	60	1.00×	4	14	0.86×	3	55	1.05×	14	1.48×
Codex	47	0.97×	13	5	1.04×	13	43	0.94×	5	1.35×

K End-to-End Results per Model

Qwen3-VL-235B uses 200 requests per throughput workload and 32 correctness samples; GLA and OpenFold3 were re-run on cloud B200 nodes after a harness fix, and Codex's sets on a separate B200 node (its Llama baseline throughput is within 0.02% of the other node's). Every set's speedups and discrepancies are relative to the baseline measured in the same run. OpenFold3 runs eagerly and deterministically, so its noise run reproduces the reference bit for bit while its diffusion sampling amplifies any numerical change; its g_i therefore comes from a calibration run that instead disables cuBLAS reduced-precision bf16 reductions, a benign numerical change. Table 13 counts the kernels each set requests and keeps across models, and why kernels were dropped; *Ind. + Seq.* pools an agent's two sets. Table 14 lists, for every model and agent set, the outcome of the all-winners run and of the deployable subset, the number of dropped kernels, the throughput and latency speedups and the mean correctness discrepancy $\bar{d}$, next to the model's noise floor ($\bar{d}$ of the noise run and the calibrated g_i). Table 15 reports the agent ranking under $0.5\times$–$2\times$ scaling of the correctness thresholds g_i and f_i.

Table 13: **Why kernel-level winners are dropped end to end.** Kernels are counted per (model, kernel) pair. Causes: *Comp.*: breaks the compilation stack (not traceable by `torch.compile`, or breaks dynamic shapes); *Run.*: crashes at run time on real engine shapes; *T/O*: hangs; *Imp.*: fails to import; *Inc.*: runs but produces wrong outputs. *AW crash / broken*: models where the full winner set crashes / runs with wrong outputs; *Not conv.*: no deployable subset found within the attempt budget.

		Kernels		**Dropped, by cause**					**Models**				
Agent	Mode	Req.	Kept	Comp.	Run.	T/O	Imp.	Inc.	AW crash	AW broken	Empty	Not conv.	N/A
Dr. Kernel	Independent	53	31	6	7	0	0	4	6	3	0	1	0
	Sequential	54	25	6	7	0	0	5	6	3	0	2	0
	Ind. + Seq.	107	56	12	14	0	0	9	12	6	0	3	0
Claude Code	Independent	175	85	11	11	1	0	2	6	3	0	4	0
	Sequential	190	93	12	8	1	0	3	8	2	0	4	0
	Ind. + Seq.	365	178	23	19	2	0	5	14	5	0	8	0
KDA	Independent	171	160	2	9	0	0	0	7	0	0	0	0
	Sequential	193	178	9	5	0	1	0	5	0	0	0	0
	Ind. + Seq.	364	338	11	14	0	1	0	12	0	0	0	0
AKO	Independent	190	117	1	14	1	0	3	7	2	0	3	0
	Sequential	201	122	14	19	0	1	2	8	1	0	2	0
	Ind. + Seq.	391	239	15	33	1	1	5	15	3	0	5	0
Codex	Independent	153	33	9	23	0	0	4	10	0	0	6	0
	Sequential	180	96	14	24	0	2	4	9	1	0	2	0
	Ind. + Seq.	333	129	23	47	0	2	8	19	1	0	8	0

Table 14: **End-to-end results per model.** AW / Dep.: status of the all-winners run and of the deployable subset (n.c.: no deployable subset within the attempt budget); Drop: dropped kernels; s_{thr}, s_{lat}: throughput and latency speedup of the deployable subset over the production baseline; $\bar{d}$: mean per-sample discrepancy; ‡: g_i from the benign-numerics calibration run instead of the noise run.

	Noise			Independent					Sequential				
Model	$\bar{d}$	g_i	Agent	AW / Dep.	Drop	s_{thr}	s_{lat}	$\bar{d}$	AW / Dep.	Drop	s_{thr}	s_{lat}	$\bar{d}$
Llama-3.1-8B	0.005	0.018	Dr. Kernel	crash / ok	3	0.96	0.99	0.000	crash / ok	3	1.01	1.00	0.000
			Claude Code	crash / n.c.	3	—	—	—	crash / n.c.	4	—	—	—
			KDA	crash / ok	2	1.02	1.02	0.002	crash / ok	6	1.01	1.01	0.000
			AKO	crash / ok	2	1.02	1.02	0.006	crash / ok	6	1.02	1.02	0.000
			Codex	crash / ok	2	0.98	1.00	0.004	crash / ok	3	1.01	1.02	0.004
GPT-OSS-120B	0.000	0.001	Dr. Kernel	crash / ok	2	1.01	0.98	0.000	crash / ok	2	1.01	0.98	0.000
			Claude Code	crash / ok	3	1.03	0.99	0.014	crash / ok	3	1.02	1.01	0.021
			KDA	crash / ok	3	1.02	0.99	0.014	crash / ok	3	1.01	1.03	0.014
			AKO	crash / n.c.	3	—	—	—	crash / ok	7	1.00	0.93	0.012
			Codex	crash / ok	4	0.98	1.02	0.000	crash / ok	5	0.99	1.02	0.000
Qwen3-Next-80B	0.000	0.001	Dr. Kernel	broken / ok	1	1.02	0.99	0.021	broken / ok	1	1.02	0.99	0.021
			Claude Code	crash / n.c.	3	—	—	—	crash / ok	1	1.02	1.03	0.019
			KDA	crash / ok	1	1.02	1.00	0.019	ok / ok	0	1.03	1.00	0.017
			AKO	crash / ok	3	1.03	1.01	0.019	crash / ok	4	1.05	1.08	0.019
			Codex	crash / ok	3	1.01	1.04	0.019	crash / ok	2	1.15	1.06	0.356†
Kimi-Linear-48B	0.013	0.041	Dr. Kernel	crash / ok	2	1.01	0.97	0.008	crash / ok	2	0.99	0.96	0.000
			Claude Code	crash / ok	5	1.01	0.97	0.024	crash / ok	5	1.01	1.02	0.024
			KDA	crash / ok	1	1.01	0.97	0.018	crash / ok	3	1.03	1.00	0.018
			AKO	crash / ok	3	0.99	1.08	0.023	crash / n.c.	5	—	—	—
			Codex	crash / n.c.	5	—	—	—	crash / ok	9	0.99	0.97	0.013
GLA-2.7B	0.000	0.001	Dr. Kernel	broken / ok	1	0.90	0.84	0.001	broken / ok	2	0.91	0.82	0.001
			Claude Code	broken / ok	1	1.27	1.70	0.001	broken / ok	2	1.27	1.75	0.001
			KDA	ok / ok	0	1.11	1.23	0.001	ok / ok	0	1.11	1.15	0.001
			AKO	broken / ok	2	1.14	1.18	0.001	broken / ok	3	1.17	1.31	0.004
			Codex	crash / ok	2	0.74	0.58	0.009	crash / ok	2	0.76	0.60	0.005
FLUX.1-dev	0.000	0.001	Dr. Kernel	broken / ok	1	1.01	0.92	0.013	broken / n.c.	1	—	—	—
			Claude Code	broken / ok	1	1.16	1.31	0.010	broken / ok	1	1.16	1.47	0.012
			KDA	ok / ok	0	1.22	1.25	0.014	ok / ok	0	1.20	1.31	0.011
			AKO	ok / ok	0	1.24	1.34	0.132†	crash / ok	2	1.18	1.49	0.012
			Codex	crash / n.c.	4	—	—	—	broken / n.c.	2	—	—	—
Qwen3-VL-235B	0.025	0.045	Dr. Kernel	crash / ok	3	0.80	1.00	0.090†	crash / n.c.	3	—	—	—
			Claude Code	crash / n.c.	3	—	—	—	crash / n.c.	4	—	—	—
			KDA	crash / ok	1	1.13	1.08	0.024	crash / ok	1	1.14	1.14	0.026
			AKO	crash / n.c.	1	—	—	—	crash / n.c.	3	—	—	—
			Codex	crash / n.c.	5	—	—	—	crash / n.c.	5	—	—	—
YOLOv10n	0.000	0.001	Dr. Kernel	crash / n.c.	2	—	—	—	crash / ok	1	1.08	0.97	0.000
			Claude Code	broken / n.c.	0	—	—	—	crash / n.c.	2	—	—	—
			KDA	crash / ok	2	1.02	1.26	0.003	crash / ok	2	1.19	0.86	0.002
			AKO	crash / ok	4	1.32	1.37	0.003	ok / ok	0	1.32	3.56	0.003
			Codex	crash / n.c.	5	—	—	—	crash / ok	6	1.15	1.25	0.040
OpenFold3	0.000	0.157‡	Dr. Kernel	ok / ok	0	1.06	1.03	0.121	ok / ok	0	1.04	1.00	0.121
			Claude Code	crash / ok	6	1.40	1.45	0.207†	crash / n.c.	1	—	—	—
			KDA	ok / ok	0	1.07	1.66	0.112	ok / ok	0	0.62	0.84	0.129
			AKO	crash / ok	1	1.27	1.72	0.113	crash / ok	5	1.25	1.48	0.130
			Codex	crash / n.c.	3	—	—	—	crash / ok	6	1.65	1.81	0.157
BGE-M3	0.000	0.001	Dr. Kernel	ok / ok	0	1.03	1.03	0.000	ok / ok	0	1.03	1.04	0.000
			Claude Code	ok / ok	0	1.07	1.07	0.000	ok / ok	0	1.05	1.09	0.000
			KDA	ok / ok	0	1.11	1.10	0.000	ok / ok	0	1.07	1.07	0.000
			AKO	ok / ok	0	1.03	0.81	0.000	ok / ok	0	0.96	1.09	0.000
			Codex	ok / ok	0	1.09	1.05	0.005	ok / ok	0	1.07	1.08	0.005
Oasis-500M	0.000	0.001	Dr. Kernel	crash / ok	2	1.03	0.84	0.003	crash / ok	3	1.01	0.85	0.003
			Claude Code	ok / ok	0	1.02	0.93	0.035	crash / ok	1	1.13	0.96	0.034
			KDA	crash / ok	1	1.01	1.26	0.040	ok / ok	0	1.14	1.22	0.039
			AKO	broken / n.c.	0	—	—	—	crash / ok	1	0.81	1.28	0.036
			Codex	crash / n.c.	3	—	—	—	crash / ok	4	1.07	0.84	0.026

Table 15: **Sensitivity of the MACROEVAL ranking** to scaling the correctness thresholds g_i and f_i.

		$(0.5\times)\, g_i, f_i$		$(1\times)\, g_i, f_i$		$(2\times)\, g_i, f_i$	
Agent	Mode	Score	Rank	Score	Rank	Score	Rank
Dr. Kernel	Independent	0.48	4/10	0.63	4/10	0.74	4/10
	Sequential	0.44	5/10	0.55	5/10	0.56	6/10
	Ind. + Seq.	0.46	2/5	0.59	3/5	0.64	2/5
Claude Code	Independent	0.24	8/10	0.35	8/10	0.47	7/10
	Sequential	0.26	7/10	0.34	9/10	0.35	9/10
	Ind. + Seq.	0.25	4/5	0.35	4/5	0.41	4/5
KDA	Independent	0.78	1/10	1.11	1/10	1.12	1/10
	Sequential	0.76	2/10	1.03	2/10	1.04	2/10
	Ind. + Seq.	0.77	1/5	1.07	1/5	1.08	1/5
AKO	Independent	0.28	6/10	0.42	7/10	0.43	8/10
	Sequential	0.57	3/10	0.85	3/10	0.86	3/10
	Ind. + Seq.	0.41	3/5	0.61	2/5	0.62	3/5
Codex	Independent	0.10	10/10	0.11	10/10	0.11	10/10
	Sequential	0.20	9/10	0.55	6/10	0.56	5/10
	Ind. + Seq.	0.15	5/5	0.30	5/5	0.30	5/5

L LEADERBOARD AND SUBMISSION PORTAL

Figure 8 shows the leaderboard (the ranking of Table 2) and Figure 9 the submission portal.

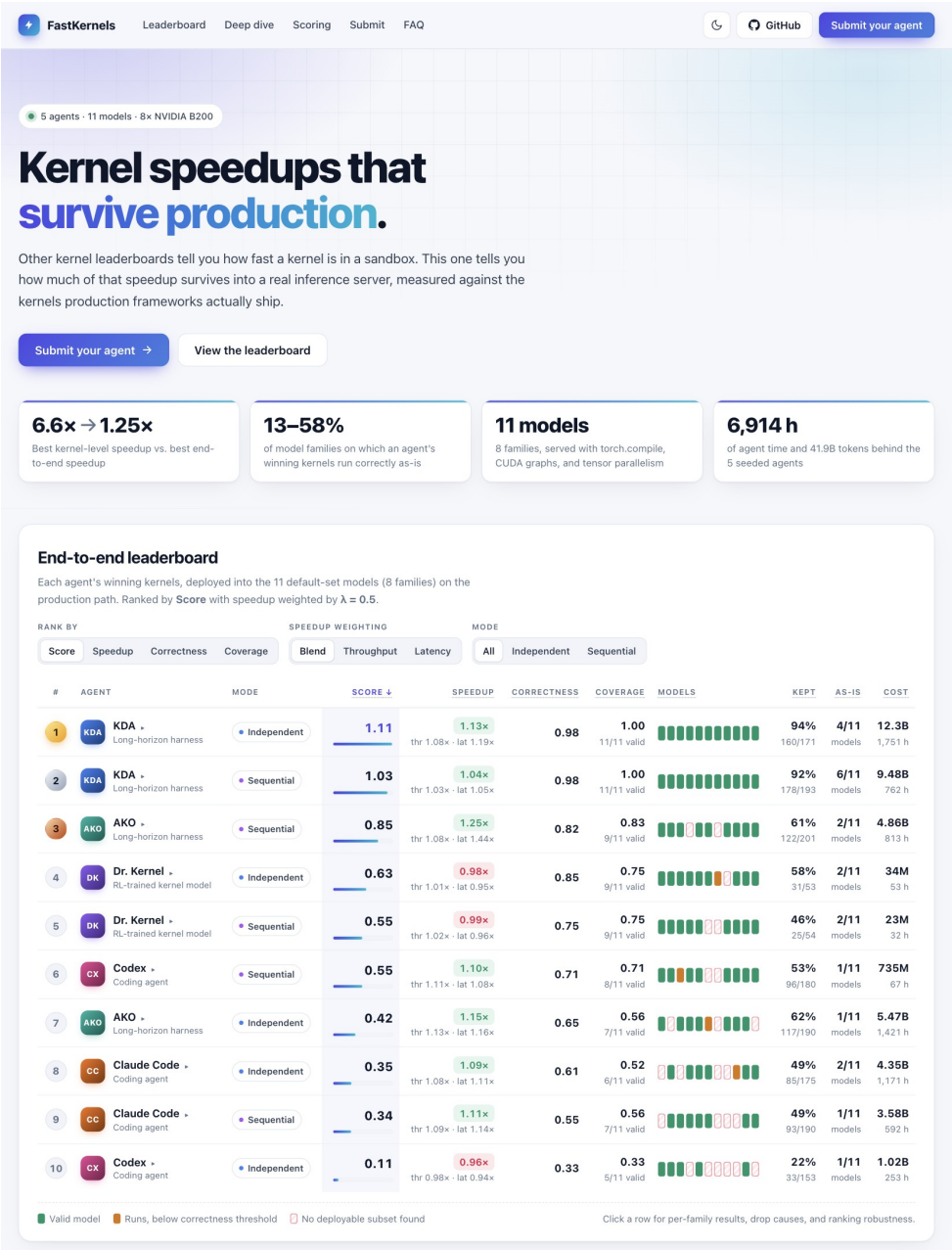


Figure 8: **Leaderboard.**

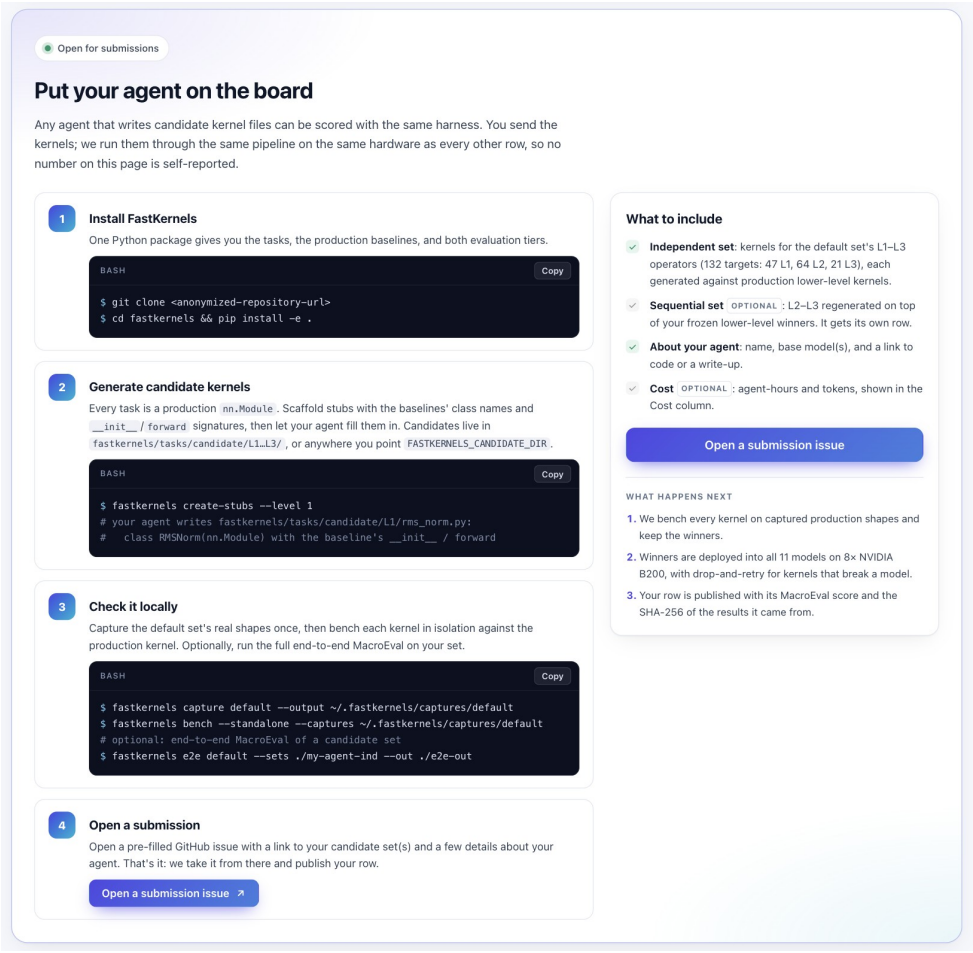


Figure 9: **Submission portal.** Repository URL anonymized for review.

M HUGGINGFACE TRANSFORMERS COVERAGE

We measure coverage empirically, by implementing every architecture in HuggingFace Transformers (commit `89b6b17`) on FASTKERNELS. For each architecture, an agent implemented the model on top of the FASTKERNELS repository using only its existing operators: minimal patches were allowed, but no new operators or primitives. To check correctness, we ran a few forward passes on a fixed batch whose input sizes exercise the model's relevant computation (e.g., for models with windowed attention, a sequence longer than the window): one step for non-autoregressive, embedding, and prefill-only models, and three steps (one prefill, two decodes) for autoregressive models. We then compared the output of the FASTKERNELS implementation with that of native HF and consider the architecture supported if the implementation runs without errors and the relative L2 difference is within a `dtype`-dependent threshold: 0.01% (FP32), 0.5% (FP16), 2% (BF16), 5% (FP8), and 10–20% (FP4); mixed-precision models use the threshold of their lowest-precision format. Architectures without a public checkpoint use the constructor defaults of their HF configuration with random weights.

Table 16 lists 488 architectures: 468 are within their threshold, 17 run but exceed it (marked †), and 4 (marked *) are flagged by the HF audit only because the HF implementation differs from the version in the reference library FASTKERNELS pins (vLLM for `kimi_linear`; FLA's RWKV-7 for `rwkv`; the SAM 3 release for `sam3_video`; timm for `timm_wrapper`); we count these as supported because FASTKERNELS supports a valid version of each. The remaining architectures fall in three groups. *Requiring new operations* (5): `autoformer`, `informer`, `rag`, `time_series_transformer`, `xlstm`. *HF reference issues* (6), where the native HF implementation itself fails, so no reference output exists: `deepseek_v4`, `gemma4_assistant`, `granite4_vision`, `minicpmv4_7`, `nllb_moe`, `sam_hq`. *Excluded* (5): generic wrappers that compose other architectures and have no modeling code of their own: `encoder_decoder`, `speech_encoder_decoder`, `timm_backbone`, `vision_encoder_decoder`, `vision_text_dual_encoder`. Coverage is therefore **94.6% (472/499)**: excluding the wrappers, the architectures above their threshold (unless marked *), those that require new operations, and those without a working HF reference count as unsupported.

Table 16: **HuggingFace Transformers coverage audit** (commit `89b6b17`): every architecture implemented on FASTKERNELS and the relative L2 difference (%) of its output from native HF. †: above the threshold for its `dtype`. *: flagged only because HF differs from the version FASTKERNELS pins; counted as supported.

Architecture	HF model	dtype	Rel. L2 (%)
afmoe	arcee-ai/Trinity-Mini	bf16	1.36
aimv2	apple/aimv2-large-patch14-224-lit	bf16	0
albert	albert/albert-base-v2	bf16	0.871
align	kakaobrain/align-base	bf16	0
altclip	BAAI/AltCLIP	bf16	0
apertus	swiss-ai/Apertus-8B-Instruct-2509	bf16	0
arcee	arcee-ai/AFM-4.5B	bf16	0.832
aria	Rhymes-AI/Aria	bf16	0.518
audio_spectrogram_transformer	Constructor defaults	bf16	0
audioflamingo3	nvidia/audio-flamingo-3-hf	bf16	0.639
axk1	skt/A.X-K1	bf16	0.395
axk2	Constructor defaults	bf16	0.171
aya_vision	unsloth/aya-vision-8b	bf16	0
bamba	ibm-ai-platform/Bamba-9B-v2	bf16	0
bark	suno/bark	bf16	0.913
bart	facebook/bart-large-cnn	bf16	0
beit	Constructor defaults	bf16	0
bert	google-bert/bert-base-uncased	bf16	0
bert_generation	google/bert_for_seq_generation_L-24_bbc_encoder	bf16	0
big_bird	google/bigbird-roberta-base	bf16	0
bigbird_pegasus	google/bigbird-pegasus-large-arxiv	bf16	0
biogpt	microsoft/biogpt	bf16	0
bit	Constructor defaults	bf16	0
bitnet	microsoft/bitnet-b1.58-2B-4T	bf16	0
blenderbot	facebook/blenderbot-400M-distill	bf16	0
blenderbot_small	facebook/blenderbot_small-90M	bf16	0
blip	Salesforce/blip-image-captioning-base	bf16	0
blip_2	Salesforce/blip2-opt-2.7b	bf16	0

Table 16 (continued)

Architecture	HF model	dtype	Rel. L2 (%)
blip_text	Constructor defaults	bf16	0
bloom	bigscience/bloom	bf16	0
blt	itazap/blt-1b-hf	bf16	0
bridgetower	BridgeTower/bridgetower-base	bf16	0
bros	jinho8345/bros-base-uncased	bf16	0
camembert	almanach/camembert-base	bf16	0
canary	nvidia/canary-1b-v2	bf16	0
canine	Constructor defaults	fp32	1.95×10^{-4}
chameleon	nopperl/chameleon-7b-hf	bf16	0.465
chinese_clip	OFA-Sys/chinese-clip-vit-base-patch16	bf16	0
chmv2	Constructor defaults	bf16	6.32×10^{-6}
clap	laion/clap-htsat-unfused	bf16	0
clip	openai/clip-vit-base-patch32	bf16	0
clipseg	CIDAS/clipseg-rd64-refined	bf16	1.35
clvp	susnato/clvp_dev	fp32	0
codegen	Salesforce/codegen-350M-mono	fp16	0
cohere	Constructor defaults	bf16	0
cohere2	Constructor defaults	bf16	0
cohere2_moe	CohereLabs/command-a-plus-05-2026	bf16	0.528
cohere2_vision	mlx-community/command-a-vision-07-2025-4bit	bf16	0.638
cohere_asr	Constructor defaults	bf16	0
cohere_compass	CohereLabs/North-Micro-Vision-Instruct	bf16	0.789
colmodernvbert	ModernVBERT/colmodernvbert-merged + ColModernVBertConfig wrapper	bf16	0
colpali	vidore/colpaligemma-3b-pt-448-base + ColPaliConfig wrapper	bf16	1.01
colqwen2	vidore/colqwen2-v1.0-hf	bf16	0.829
conditional_detr	microsoft/conditional-detr-resnet-50	bf16	0
convbert	YituTech/conv-bert-base	bf16	0
convnext	Constructor defaults	bf16	0
convnextv2	Constructor defaults	bf16	0
cosmos3_edge	nvidia/Cosmos3-Edge	bf16	0
cosmos3_omni	nvidia/Cosmos3-Nano	bf16	1.2
cpmant	openbmb/cpm-ant-10b	bf16	0
csm	Constructor defaults	bf16	0.00206
ctrl	Salesforce/ctrl	bf16	0.366
cvt	Constructor defaults	bf16	0
cwm	Constructor defaults	bf16	0.884
d_fine	ustc-community/dfine-xlarge-coco	bf16	0
dab_detr	IDEA-Research/dab-detr-resnet-50	fp32	0
dac	descript/dac_16khz	bf16	0
data2vec_audio	Constructor defaults	bf16	0
data2vec_text	facebook/data2vec-text-base	bf16	0
data2vec_vision	Constructor defaults	bf16	0
dbrx	alpindale/dbrx-instruct (mirror)	bf16	0
deberta	microsoft/deberta-base	fp32	1.64×10^{-4}
deberta_v2	microsoft/deberta-v2-xlarge	bf16	0
decision_transformer	edbeeching/decision-transformer-gym-hopper-medium	bf16	0
deepseek_ocr2	deepseek-community/DeepSeek-OCR-2	bf16	1.77
deepseek_v2	deepseek-ai/DeepSeek-V2-Lite	bf16	1
deepseek_v3	deepseek-ai/DeepSeek-V3	fp8/bf16	0
deepseek_v32	Constructor defaults	bf16	1.64
deepseek_vl	deepseek-community/deepseek-vl-1.3b-chat	fp16	0.0769
deepseek_vl_hybrid	deepseek-community/deepseek-vl-7b-chat	fp16	0.0693
deformable_detr	SenseTime/deformable-detr	bf16	0
deimv2	harshaljanjani/DEIMv2_HGNetv2_N_COCO_Transformers	bf16	0
deit	Constructor defaults	bf16	0
depth_anything	LiheYoung/depth-anything-small-hf	bf16	0.141
depth_pro	apple/DepthPro-hf	fp16	0.121
detr	facebook/detr-resnet-50-panoptic	bf16	0
dia	Constructor defaults	bf16	0.537
diffllama	kajuma/DiffLlama-0.3B-handcut	bf16	0.746
diffusion_gemma	google/diffusiongemma-26B-A4B-it	bf16	81[†]
dinat	Constructor defaults	bf16	0
dinov2	Constructor defaults	bf16	0
dinov2_with_registers	Constructor defaults	bf16	0
dinov3_convnext	Constructor defaults	bf16	0
dinov3_vit	Constructor defaults	bf16	0
distilbert	distilbert/distilbert-base-uncased	bf16	0.855
doge	Constructor defaults	bf16	0
donut_swin	Constructor defaults	bf16	0
dots1	rednote-hilab/dots.llm1.base	bf16	0.239
dpr	facebook/dpr-question_encoder-single-nq-base	bf16	0.972
dpt	Constructor defaults	bf16	0

Table 16 (continued)

Architecture	HF model	dtype	Rel. L2 (%)
edgetam	yonigozlan/EdgeTAM-hf	bf16	0
edgetam_video	yonigozlan/EdgeTAM-hf	bf16	0
efficientloftr	zju-community/efficientloftr	bf16	0.0292
efficientnet	Constructor defaults	bf16	0
electra	google/electra-small-generator	bf16	0.885
emu3	BAAI/Emu3-Chat-hf	bf16	0
encodec	facebook/encodec_24khz	bf16	0.0579
eomt	tue-mps/coco_panoptic_eomt_large_640	bf16	0
eomt_dinov3	Constructor defaults	bf16	0
ernie	nghuyong/ernie-1.0-base-zh	bf16	0
ernie4_5	baidu/ERNIE-4.5-0.3B-PT	bf16	0
ernie4_5_moe	baidu/ERNIE-4.5-21B-A3B-PT	bf16	0.542
ernie4_5_vl_moe	baidu/ERNIE-4.5-VL-28B-A3B-PT	bf16	0.42
esm	facebook/esm-1b	bf16	1.73
esmc	biohub/ESMC-6B-hf	bf16	0
esmfold	facebook/esmfold_v1	bf16	4.80×10^{-6}
esmfold2	biohub/ESMFold2-hf	fp32	2.37×10^{-5}
eurobert	EuroBERT/EuroBERT-210m	bf16	0
evolla	westlake-repl/Evolla-10B-DPO-hf	bf16	0
exaone4	LGAI-EXAONE/EXAONE-4.0-32B	bf16	0
exaone4_5	LGAI-EXAONE/EXAONE-4.5-33B	bf16	1.27
exaone_moe	LGAI-EXAONE/K-EXAONE-236B-A23B	bf16	1.65
falcon	tiiuae/falcon-7b-instruct	bf16	1.45
falcon_h1	tiiuae/Falcon-H1-7B-Instruct	bf16	1.25
falcon_mamba	tiiuae/falcon-mamba-7b-instruct	bf16	0
fast_vlm	KamilaMila/FastVLM-0.5B	bf16	0.69
fastspeech2_conformer	espnet/fastspeech2_conformer_with_hifigan	fp32	1.91×10^{-4}
flaubert	flaubert/flaubert_base_uncased	bf16	1.09
flava	facebook/flava-full	bf16	0
flex_olmo	allenai/FlexOlmo-7x7B-1T	bf16	0.86
florence2	florence-community/Florence-2-large	fp16	0
fnet	google/fnet-base	fp32	1.70×10^{-4}
focalnet	Constructor defaults	bf16	0
fsmt	facebook/wmt19-ru-en	bf16	0
fun_asr_nano	FunAudioLLM/Fun-ASR-Nano-2512-hf	bf16	0
funnel	funnel-transformer/small	bf16	0
fuyu	adept/fuyu-8b	bf16	0
gemma	Constructor defaults	bf16	0.451
gemma2	Constructor defaults	bf16	0.389
gemma3	Pinned Gemma3 4B recipe	bf16	0
gemma3n	Constructor defaults	bf16	0
gemma4	google/gemma-4-e2b-it	bf16	0
gemma4_unified	google/gemma-4-12B-it	bf16	5.88[†]
gemma4_unified_assistant	google/gemma-4-e2b-it-assistant	bf16	0
git	microsoft/git-base-coco	bf16	0
glm	THUDM/glm-4-9b-chat	bf16	0.905
glm4	Constructor defaults	bf16	0.761
glm46v	zai-org/GLM-4.1V-9B-Thinking	bf16	0
glm4_moe	zai-org/GLM-4.5	bf16	0.212
glm4_moe_lite	zai-org/GLM-4.7-Flash	bf16	0.99
glm4v	zai-org/GLM-4.1V-9B-Thinking	bf16	0
glm4v_moe	zai-org/GLM-4.5V	bf16	0.143
glm5_next	zai-org/GLM-5.3-Flash	fp8/bf16	19.5[†]
glm_image	zai-org/GLM-Image	bf16	0
glm_moe_dsa	zai-org/GLM-5	bf16	1.92
glm_ocr	zai-org/GLM-OCR	bf16	0
glmasr	zai-org/GLM-ASR-Nano-2512	bf16	0
glpn	vinvino02/glpn-kitti	bf16	0.062
got_ocr2	stepfun-ai/GOT-OCR-2.0-hf	bf16	0
gpt2	openai-community/gpt2	bf16	0
gpt_bigcode	bigcode/gpt_bigcode-santacoder	bf16	0
gpt_neo	EleutherAI/gpt-neo-1.3B	bf16	0
gpt_neox	EleutherAI/gpt-neox-20b	fp16	0.0522
gpt_neox_japanese	abeja/gpt-neox-japanese-2.7b	bf16	0.687
gpt_oss	openai/gpt-oss-20b	mxfp4/bf16	0
gptj	EleutherAI/gpt-j-6B	bf16	0
granite	ibm-granite/granite-3.0-8b-base	bf16	0.275
granite_speech	ibm-granite/granite-speech-3.3-2b	bf16	0
granite_speech5	ibm-granite/granite-speech-5.0-470m-turbo ctc	bf16	0
granite_speech_plus	ibm-granite/granite-speech-4.1-2b-plus	bf16	0
granite_swa	ibm-granite/granite-swash-2b	bf16	0
granitemoe	ibm/PowerMoE-3b	bf16	1.69
granitemoe_swa	ibm-granite/granite-swash-3b-a600m	bf16	9.34[†]
granitemoehybrid	ibm-granite/granite-4.0-h-tiny	bf16	0
granitemoeshared	ibm-research/moe-7b-1b-active-shared-exp erts	bf16	0

Table 16 (continued)

Architecture	HF model	dtype	Rel. L2 (%)
grounding_dino	IDEA-Research/grounding-dino-tiny	fp32	6.4×10^{-5}
groupvit	nvidia/groupvit-gcc-yfcc	bf16	0
helium	kyutai/helium-1-preview	bf16	0
hgnet_v2	Constructor defaults	bf16	0
hiera	Constructor defaults	bf16	0
higgs_audio_v2	eustlb/higgs-audio-v2-generation-3B-base	bf16	0
higgs_audio_v2_tokenizer	eustlb/higgs-audio-v2-tokenizer	bf16	0
hrm_text	sapientinc/HRM-Text-1B	bf16	0
hubert	facebook/hubert-large-ls960-ft	bf16	0
hunyuan_v1_dense	tencent/Hunyuan-0.5B-Pretrain	bf16	0
hunyuan_v1_moe	tencent/Hunyuan-A13B-Instruct	bf16	0
hunyuan_vl	tencent/HunyuanOCR	bf16	0.724
hy_v3	tencent/Hy3-preview	bf16	0.712
hy_v4	tencent/Hy4-preview	bf16	2.55†
hyperclovax	naver-hyperclovax/HyperCLOVAX-SEED-Think-14B	bf16	0
hyperclovax_vision_v2	naver-hyperclovax/HyperCLOVAX-SEED-Think-32B	bf16	0.78
ibert	kssteven/ibert-roberta-base	bf16	1.33
idefics	HuggingFaceM4/idefics-9b	bf16	0.651
idefics2	HuggingFaceM4/idefics2-8b-base	bf16	0.749
idefics3	HuggingFaceM4/Idefics3-8B-Llama3	bf16	1.12
ijepa	Constructor defaults	bf16	0
imagegpt	openai/imagegpt-small	bf16	0
inkling	thinkingmachines/Inkling	bf16	4.15†
instructblip	Salesforce/instructblip-vicuna-7b	bf16	0.539
instructblipvideo	Salesforce/instructblip-vicuna-7b	bf16	0.483
internvl	OpenGVLab/InternVL3-1B-hf	bf16	0
jais2	Constructor defaults	bf16	0
jamba	ai21labs/Jamba-v0.1	bf16	1.02
janus	deepseek-community/Janus-Pro-1B	bf16	0
jetmoe	jetmoe/jetmoe-8b	bf16	0.341
jina_embeddings_v3	jinaai/jina-embeddings-v3-hf	bf16	1.5
kimi_k25	moonshotai/Kimi-K2.6	int4/bf16	7.61
kimi_linear*	moonshotai/Kimi-Linear-48B-A3B-Instruct	bf16	16.9†
kosmos2	microsoft/kosmos-2-patch14-224	bf16	0
kosmos2_5	microsoft/kosmos-2.5	bf16	0
kyutai_speech_to_text	kyutai/stt-2.6b-en-trfs	bf16	0
laguna	poolside/laguna-XS.2	bf16	0
lasr	Constructor defaults	bf16	0.727
layoutlm	microsoft/layoutlm-base-uncased	bf16	0
layoutlmv2	microsoft/layoutlmv2-base-uncased	bf16	0
layoutlmv3	microsoft/layoutlmv3-base	bf16	0
led	allenai/led-large-16384-arxiv	bf16	0
levit	Constructor defaults	bf16	0
lfm2	LiquidAI/LFM2-1.2B	bf16	1.83
lfm2_moe	LiquidAI/LFM2-8B-A1B	bf16	1.45
lfm2_vl	LiquidAI/LFM2-VL-1.6B	bf16	0.787
lightglue	ETH-CVG/lightglue_superpoint	bf16	0
lighton_ocr	lightonai/LightOnOCR-1B-1025	bf16	0
lilt	SCUT-DLVCLab/lilt-roberta-en-base	bf16	0
llama	Constructor defaults	bf16	0.746
llama4	Constructor defaults	bf16	0
llava	llava-hf/llava-1.5-7b-hf	fp16	0.0932
llava_next	llava-hf/llava-v1.6-mistral-7b-hf	fp16	0.0961
llava_next_video	llava-hf/LLaVA-NeXT-Video-7B-hf	bf16	1.34
llava_onevision	llava-hf/llava-onevision-qwen2-7b-ov-hf	fp16	0.0756
longcat_flash	meituan-longcat/LongCat-Flash-Chat	bf16	0.725
longformer	allenai/longformer-base-4096	bf16	0
longt5	Stancld/longt5-tglobal-large-16384-pubmed-3k_steps	bf16	0
luke	studio-ousia/luke-base	bf16	0
lw_detr	AnnaZhang/lwdetr_small_60e_coco	bf16	0
lxmert	unc-nlp/lxmert-base-uncased	bf16	0
m2m_100	facebook/m2m100_418M	bf16	0
mamba	state-spaces/mamba-130m-hf	bf16	0
mamba2	mistralai/Mamba-Codestral-7B-v0.1	bf16	0
marian	Helsinki-NLP/opus-mt-en-de	bf16	0
markuplm	microsoft/markuplm-base	fp16	0
mask2former	facebook/mask2former-swin-small-coco-instance	bf16	0
maskformer	facebook/maskformer-swin-base-ade	fp32	0
maskformer_swin	Constructor defaults	fp32	2.82×10^{-5}
mbart	facebook/mbart-large-en-ro	bf16	0
megatron_bert	Constructor defaults	bf16	0
mellum	JetBrains/Mellum2-12B-A2.5B-Base	bf16	0.817

Table 16 (continued)

Architecture	HF model	dtype	Rel. L2 (%)
metaclip_2	facebook/metaclip-2-worldwide-huge-quickgelu	bf16	0
mgp_str	alibaba-damo/mgp-str-base	bf16	0.365
mimi	kyutai/mimi	bf16	0
mimo_v2_flash	XiaomiMiMo/MiMo-V2-Flash	fp8/bf16	12.2†
minicpm3	openbmb/MiniCPM3-4B	bf16	0
minicpmv4_6	openbmb/MiniCPM-V-4_6	bf16	1.27
minimax	MiniMaxAI/MiniMax-Text-01-hf	bf16	0.255
minimax_m2	MiniMaxAI/MiniMax-M2	fp8/bf16	8.69×10^{-5}
minimax_m3_vl	MiniMaxAI/MiniMax-M3	bf16	3.4†
ministral	mistralai/Ministral-8B-Instruct-2410	bf16	0.768
ministral3	mistralai/Ministral-3-8B-Base-2512	bf16	0.649
mistral	mistralai/Mistral-7B-v0.1	bf16	0.81
mistral3	mistralai/Mistral-Small-3.1-24B-Instruct-2503	bf16	0
mistral4	mistralai/Mistral-Small-4-119B-2603	fp8/bf16/fp32	0
mixtral	mistralai/Mixtral-8x7B-v0.1	bf16	0
mlcd	DeepGlint-AI/mlcd-vit-bigG-patch14-448	bf16	1.24
mllama	Constructor defaults	bf16	0
mm_grounding_dino	openmmlab-community/mm_grounding_dino_tiny_o365v1_goldg_v3det	fp32	4.4×10^{-5}
mobilebert	google/mobilebert-uncased	bf16	0
mobilenet_v1	Constructor defaults	bf16	0
mobilenet_v2	Constructor defaults	bf16	0
mobilevit	Constructor defaults	bf16	0
mobilevitv2	Constructor defaults	bf16	0.701
modernbert	answerdotai/ModernBERT-base	bf16	0
modernbert_decoder	blab-jhu/test-32m-dec	bf16	0
modernvbert	ModernVBERT/modernvbert	bf16	1.3
moonshine	UsefulSensors/moonshine-tiny	bf16	0
moonshine_streaming	UsefulSensors/moonshine-streaming-tiny	bf16	0
moshi	kmhf/hf-moshiko	bf16	0
mpnet	microsoft/mpnet-base	bf16	0
mpt	Constructor defaults	bf16	0
mra	uw-madison/mra-base-512-4	fp32	4.1×10^{-5}
mt5	google/mt5-small	bf16	0
muse_glimmer	meta-models/Muse-Glimmer-30B	bf16	12.2†
muse_glimmer_assistant	meta-models/Muse-Glimmer-30B-assistant	bf16	0.413
musicflamingo	nvidia/music-flamingo-2601-hf	bf16	0
musicgen	facebook/musicgen-small	bf16	0
musicgen_melody	facebook/musicgen-melody	bf16	0
mvp	RUCAIBox/mvp	bf16	0
nanochat	Constructor defaults	bf16	0
nemotron	nvidia/Minitron-4B-Base	bf16	0.843
nemotron3_5_asr	nvidia/nemotron-3.5-asr-streaming-0.6b	bf16	0.0547
nemotron3_diarization	nvidia/Nemotron-3-Diarization	bf16	0
nemotron_asr_streaming	nvidia/nemotron-speech-streaming-en-0.6b	bf16	0.569
nemotron_h	nvidia/Nemotron-H-8B-Reasoning-128K	bf16	1.53
nemotron_h_omni	nvidia/Nemotron-3-Nano-Omni-30B-A3B-Reasoning-BF16	bf16	13.2†
neomme	Hcompany/NeoMME-260M	bf16	0.118
neucodec	Constructor defaults	bf16	16.1†
nomic_bert	Constructor defaults	bf16	0
nystromformer	uw-madison/nystromformer-512	bf16	0
olmo	allenai/OLMo-7B-hf	bf16	0.711
olmo2	allenai/OLMo-2-0425-1B	bf16	0
olmo3	allenai/Olmo-3-7B-Instruct	bf16	0
olmo_hybrid	allenai/Olmo-Hybrid-7B	bf16	0.639
olmoe	allenai/OLMoE-1B-7B-0924	bf16	0.557
omdet_turbo	omlab/omdet-turbo-swin-tiny-hf	bf16	0
oneformer	shi-labs/oneformer_ade20k_swin_tiny	bf16	0.302
openai	openai-community/openai-gpt	bf16	0
openai_privacy_filter	openai/privacy-filter	bf16	1.3
opt	facebook/opt-350m	fp16	0
ovis2	thisisiron/Ovis2-2B-hf	bf16	0.702
owlv2	google/owlv2-base-patch16-ensemble	bf16	0
owlvit	google/owlvit-base-patch32	bf16	0
paddleocr_vl	PaddlePaddle/PaddleOCR-VL	bf16	0
paligemma	Constructor defaults	bf16	0.0153
parakeet	nvidia/parakeet-ctc-1.1b	bf16	0
patchtsmixer	ibm/patchtsmixer-etth1-pretrain	bf16	0
patchtst	namctin/patchtst_etth1_pretrain	bf16	0.132
pe_audio	facebook/pe-av-large	bf16	0
pe_audio_video	facebook/pe-av-large	bf16	0
pe_video	facebook/pe-av-large	bf16	0
pegasus	google/pegasus-xsum	bf16	0
pegasus_x	google/pegasus-x-large	bf16	0

Table 16 (continued)

Architecture	HF model	dtype	Rel. L2 (%)
perceiver	Constructor defaults	bf16	0
perception_lm	facebook/Perception-LM-1B	bf16	0
persimmon	adept/persimmon-8b-base	bf16	0
phi	microsoft/phi-1	bf16	0
phi3	microsoft/Phi-3-mini-4k-instruct	bf16	0.929
phi4_multimodal(speech)	microsoft/Phi-4-multimodal-instruct	bf16	0
phimoe	microsoft/Phi-3.5-MoE-instruct	bf16	0
pi0	Constructor defaults	bf16	0.282
pix2struct	google/pix2struct-textcaps-base	bf16	0
pixio	Constructor defaults	bf16	0
pixtral	mistral-labs/pixtral-12b	bf16	0
plbart	uclanlp/plbart-base	bf16	0
poolformer	Constructor defaults	bf16	0
pop2piano	sweetcocoa/pop2piano	bf16	0
pp_doclayout_v2	PaddlePaddle/PP-DocLayoutV2_safetensors	bf16	0
pp_doclayout_v3	PaddlePaddle/PP-DocLayoutV3_safetensors	bf16	0
pp_formulanet	PaddlePaddle/PP-FormulaNet_plus-L_safetensors	bf16	0
pp_lcnet	Constructor defaults	bf16	0
pp_lcnet_v3	Constructor defaults	bf16	0
pp_lcnet_v4	Constructor defaults	bf16	0
pp_ocrv5_mobile_det	PaddlePaddle/PP-OCRv5_mobile_det_safetensors	bf16	0
pp_ocrv5_mobile_rec	PaddlePaddle/PP-OCRv5_mobile_rec_safetensors	bf16	0
pp_ocrv5_server_det	PaddlePaddle/PP-OCRv5_server_det_safetensors	bf16	0
pp_ocrv5_server_rec	PaddlePaddle/PP-OCRv5_server_rec_safetensors	bf16	0
pp_ocrv6_medium_det	PaddlePaddle/PP-OCRv6_medium_det_safetensors	bf16	0
pp_ocrv6_small_det	PaddlePaddle/PP-OCRv6_small_det_safetensors	bf16	0
pp_ocrv6_small_rec	PaddlePaddle/PP-OCRv6_small_rec_safetensors	bf16	0
pp_ocrv6_tiny_rec	PaddlePaddle/PP-OCRv6_tiny_rec_safetensors	bf16	0
prompt_depth_anything	depth-anything/prompt-depth-anything-vits-hf	bf16	0.0479
prophetnet	microsoft/prophetnet-large-uncased	bf16	0
pvt	Constructor defaults	bf16	0
pvt_v2	Constructor defaults	bf16	0
qianfan_ocr	baidu/Qianfan-OCR	bf16	0.727
qwen2	Qwen/Qwen2-7B	bf16	0
qwen2_5_omni	Qwen/Qwen2.5-Omni-7B	bf16	0.005
qwen2_5_vl	Qwen/Qwen2.5-VL-7B-Instruct	bf16	0.789
qwen2_audio	Qwen/Qwen2-Audio-7B	bf16	0.683
qwen2_moe	Qwen/Qwen1.5-MoE-A2.7B-Chat	bf16	0.56
qwen2_vl	Qwen/Qwen2-VL-7B-Instruct	bf16	0.767
qwen3	Qwen/Qwen3-8B	bf16	0
qwen3_5	Qwen/Qwen3.5-27B	bf16	0.393
qwen3_5_moe	Qwen/Qwen3.5-35B-A3B	bf16	0.353
qwen3_asr	Qwen/Qwen3-ASR-1.7B-hf	bf16	0.827
qwen3_moe	Qwen/Qwen3-30B-A3B	bf16	0.897
qwen3_next	Qwen/Qwen3-Next-80B-A3B-Instruct	bf16	0.758
qwen3_omni_moe	Qwen/Qwen3-Omni-30B-A3B-Instruct	bf16	0
qwen3_vl	Qwen/Qwen3-VL-8B-Instruct	bf16	0.994
qwen3_vl_moe	Qwen/Qwen3-VL-30B-A3B-Instruct	bf16	0.939
qwen4_exp	Qwen/Qwen3.8-Flash-Next	bf16	12.7[†]
radio	Constructor defaults	bf16	0
recurrent_gemma	Constructor defaults	bf16	0.334
reformer	hf-internal-testing/tiny-random-reformer	fp32	8×10^{-6}
regnet	Constructor defaults	bf16	0
rembert	google/rembert	bf16	0
resnet	Constructor defaults	bf16	0
rf_detr	Roboflow/rf-detr-base	bf16	113[†]
roberta	FacebookAI/roberta-base	bf16	0
roberta_prelayernorm	andreasmadsen/efficient_mlm_m0.40	bf16	0
roc_bert	weiweishi/roc-bert-base-zh	bf16	0
roformer	junnyu/roformer_chinese_base	bf16	0
rt_detr	PekingU/rtdetr_r50vd	bf16	0
rt_detr_resnet	Constructor defaults	bf16	0
rt_detr_v2	PekingU/rtdetr_v2_r18vd	bf16	0.387
rwkv[*]	—	—	—
sam	facebook/sam-vit-base	bf16	0
sam2	danelcsb/sam2.1_hiera_tiny	bf16	0
sam2_video	facebook/sam2.1-hiera-tiny	bf16	0
sam3	Constructor defaults	bf16	0

Table 16 (continued)

Architecture	HF model	dtype	Rel. L2 (%)
sam3_lite_text	yonigozlan/sam3-litetext-s0	bf16	0
sam3_tracker	Constructor defaults	bf16	0.814
sam3_tracker_video	Constructor defaults	bf16	0
sam3_video*	—	—	—
sapiens2	facebook/sapiens2-pretrain-0.4b	bf16	0
seamless_m4t	facebook/hf-seamless-m4t-medium	bf16	0
seamless_m4t_v2	facebook/seamless-m4t-v2-large	bf16	0
seed_oss	ByteDance-Seed/Seed-OSS-36B-Instruct	bf16	0.839
segformer	nvidia/segformer-b0-finetuned-ade-512-512	bf16	0.456
seggpt	BAAI/seggpt-vit-large	bf16	0
sew	Constructor defaults	bf16	0
sew_d	Constructor defaults	bf16	0
shieldgemma2	Official ShieldGemma2 converter recipe	bf16	0
siglip	google/siglip-base-patch16-224	bf16	0.529
siglip2	Constructor defaults	bf16	0.755
slanet	PaddlePaddle/SLANet_plus_safetensors	fp32	1.02×10^{-4}
slanext	PaddlePaddle/SLANeXt_wired_safetensors	bf16	0
smollm3	HuggingFaceTB/SmolLM3-3B	bf16	0.85
smolvlm	HuggingFaceTB/SmolVLM2-2.2B-Instruct	bf16	0
solar_open	upstage/Solar-Open-100B	bf16	0.61
speech_to_text	facebook/s2t-small-librispeech-asr	bf16	0
speecht5	microsoft/speecht5_asr	bf16	0
splinter	tau/splinter-base	bf16	0
squeezebert	squeezebert/squeezebert-uncased	bf16	0.798
stablelm	stabilityai/stablelm-3b-4e1t	bf16	0
starcoder2	bigcode/starcoder2-7b	bf16	0.815
step3p7	stepfun-ai/Step-3.7-Flash	bf16	11.2†
superglue	magic-leap-community/superglue_outdoor	bf16	0
superpoint	magic-leap-community/superpoint	bf16	0
swiftformer	Constructor defaults	bf16	0
swin	Constructor defaults	bf16	0
swin2sr	Constructor defaults	bf16	1.16
swinv2	Constructor defaults	bf16	0
switch_transformers	google/switch-base-8	bf16	0
t5	google-t5/t5-small	bf16	0
t5gemma	Author T5GemmaPreset.GEMMA2_2B_2B PREFIXLM IT recipe	bf16	0.688
t5gemma2	jordimas/t5gemma-2-270m-270m	bf16	0
table_transformer	microsoft/table-transformer-detection	bf16	0
tapas	google/tapas-base	bf16	0
textnet	czczup/textnet-base	bf16	0
timesfm	google/timesfm-2.0-500m-pytorch	bf16	0
timesfm2_5	google/timesfm-2.5-200m-transformers	bf16	0
timesformer	Constructor defaults	bf16	1.21
timm_wrapper*	—	—	—
tipsv2	google/tipsv2-b14	bf16	1.86
tipsv2_dpt	google/tipsv2-b14-dpt	bf16	2.23†
trocr	Constructor defaults	bf16	0
tvp	Intel/tvp-base	bf16	0
udop	microsoft/udop-large	bf16	0
umt5	google/umt5-small	bf16	0
unispeech	Constructor defaults	bf16	0
unispeech_sat	Constructor defaults	bf16	0
univnet	dg845/univnet-dev	bf16	0
upernet	openmmlab/upernet-convnext-tiny	bf16	0.323
uvdoc	PaddlePaddle/UVDoc_safetensors	bf16	0
vaultgemma	Constructor defaults	bf16	0.621
vibevoice	vibevoice/VibeVoice-1.5B-hf	bf16	1.16
vibevoice_acoustic_tokenizer	microsoft/VibeVoice-AcousticTokenizer	bf16	0
vibevoice_asr	microsoft/VibeVoice-ASR-HF	bf16	0
video_llama_3	lkhl/VideoLLaMA3-2B-Image-HF	bf16	0.624
video_llava	LanguageBind/Video-LLaVA-7B-hf	bf16	0.716
videomae	MCG-NJU/videomae-base	bf16	0
videomt	tue-mps/videomt-dinov2-small-ytvis2019	bf16	0
videoprism	Constructor defaults	bf16	0
vilt	dandelin/vilt-b32-finetuned-coco	bf16	0
vipllava	llava-hf/vip-llava-7b-hf	fp16	0.0971
visual_bert	uclanlp/visualbert-vqa-coco-pre	bf16	0
vit	Constructor defaults	bf16	0
vit_mae	facebook/vit-mae-base	bf16	0
vit_msn	facebook/vit-msn-small	bf16	0
vitdet	Constructor defaults	bf16	0
vitmatte	hustvl/vitmatte-small-composition-1k	bf16	0
vitpose	usyd-community/vitpose-base-simple	bf16	0
vitpose_backbone	Constructor defaults	bf16	0
vits	facebook/mms-tts-eng	bf16	0

Table 16 (continued)

Architecture	HF model	dtype	Rel. L2 (%)
vivit	google/vivit-b-16x2-kinetics400	bf16	0
vjepa2	Constructor defaults	bf16	0
voxtral	mistralai/Voxtral-Mini-3B-2507	bf16	0
voxtral_realtime	mistralai/Voxtral-Mini-4B-Realtime-2602	bf16	0
wav2vec2	facebook/wav2vec2-large-960h-lv60-self	bf16	0
wav2vec2_bert	facebook/w2v-bert-2.0	bf16	0
wav2vec2_conformer	facebook/wav2vec2-conformer-rel-pos-large	bf16	0
wavlm	Constructor defaults	bf16	0
whisper	openai/whisper-tiny.en	bf16	0.795
x_clip	microsoft/xclip-base-patch32	bf16	0
xcodec	hf-audio/xcodec-hubert-librispeech	bf16	0
xcodec2	HKUSTAudio/xcodec2-hf	bf16	$10.1^{\dagger}$
xglm	facebook/xglm-564M	bf16	0
xlm	FacebookAI/xlm-mlm-en-2048	bf16	1.11
xlm_roberta	FacebookAI/xlm-roberta-base	bf16	0
xlm_roberta_xl	facebook/xlm-roberta-xl	bf16	1.98
xlnet	Constructor defaults	fp32	2.25×10^{-4}
xmod	facebook/xmod-base	bf16	1.48
yolos	hustvl/yolos-tiny	bf16	0
yoso	Constructor defaults	bf16	0
youtu	tencent/Youtu-LLM-2B	bf16	0.974
zamba	Zyphra/Zamba-7B-v1	bf16	0.512
zamba2	Constructor defaults	bf16	1.22
zaya	Zyphra/ZAYA1-8B	bf16	0.668
zoedepth	Intel/zoedepth-nyu-kitti	bf16	0.00553