JAXBench: Benchmarking Autonomous TPU Kernel Optimization

Arya Tschand [*1,2] **Charles Hong** [*3,2] **Julian Walker** [4] **Nina Cai** [2] **Shangkun Wang** [2] **Suvinay Subramanian** [2] **Sundar Dev** [2] **Vijay Janapa Reddi** [1] **Amir Yazdanbakhsh** [4] **Sethu Sankaran** [2]

Abstract

Evaluation benchmarks have driven progress in automated kernel optimization, yet existing suites target GPUs exclusively. We present JAXBENCH,[1] a TPU-native benchmark for AI-generated kernel optimization on Google Cloud TPUs. JAXBENCH comprises 50 JAX workloads, including 17 production LLM operators extracted from architectures such as Llama-3.1, DeepSeek-V3, Mixtral, Mamba-2, and AlphaFold2, and 33 fused operator sequences adapted from KernelBench. Eight of the 17 production operators ship with hand-optimized Pallas TPU kernels whose block sizes we tune via grid search, establishing reference baselines. We evaluate one-shot generation, iterative coding agents, the same iterative loop with TPU documentation injected, and a TPU-enabled Autocomp configuration augmented with the same TPU-specific documentation. With Gemini 3 Flash, best-of-N solves 13/50 at $1.01\times$ geomean and iterative refinement reaches 32/50 at $1.18\times$. Injecting TPU documentation lifts iterative refinement to 48/50 at $1.28\times$ and per-sample correctness from 5.8% to 37.3%. Autocomp solves 45/50 and converts those correct kernels into $1.36\times$ geomean with 76% beating XLA. All baselines still trail hand-tuned performance, showing that TPU kernel generation remains open. We release the benchmark, harness, and baselines for reproducible research.

1. Introduction

Efficient kernel implementations are a bottleneck in realizing the full potential of ML accelerators. Libraries such as cuBLAS, CUTLASS, and Triton (Thakkar et al., 2023; Tillet et al., 2019) and specialized kernels for GPUs (Dao et al., 2022) and TPUs (Jiang et al., 2026) have unlocked generation-over-generation performance gains, but each new model architecture, quantization scheme, and hardware revision demands fresh low-level implementations. Engineers fluent in both ML and accelerator systems programming remain in chronically short supply.

This gap has motivated progress on LLM-driven kernel generation, from one-shot completions to iterative agents with compile/profile feedback to evolutionary search (Liao et al., 2025; Tschand et al., 2025; Lange et al., 2025; Cao et al., 2026). Rigorous benchmarks have been central. KernelBench (Ouyang et al., 2025) standardized the evaluation protocol with 250 PyTorch workloads and a correctness-plus-speedup metric against CUDA. TritonBench (Li et al., 2025) targets Triton generation with hardware-aware measurement on NVIDIA and AMD GPUs. FlashInfer-Bench (Xing et al., 2026) grounds evaluation in real LLM serving traces with expert-written FlashInfer kernels as references. These benchmarks have driven progress on correctness and speedup against GPU baselines.

However, no analogous benchmark exists for TPU kernel optimization. TPUs differ fundamentally from GPUs. They are sequential machines with wide SIMD vector registers (8×128 for 32-bit values on v6e) and dedicated 256×256 systolic Matrix Multiply Units (MXUs), rather than the massively parallel SIMT execution of GPUs. Programming TPUs requires a distinct stack. JAX compiles through XLA, and kernel authoring goes through Pallas, which lowers to Mosaic rather than Triton. Pallas kernels must reason about TPU-specific concerns including VMEM/SMEM/HBM hierarchies, software pipelining with prefetch scheduling, block-shape constraints (last two dimensions divisible by 8 and 128), and lexicographic grid traversal (JAX Development Team, 2024). Pallas also appears in LLM training data orders of magnitude less frequently than CUDA or even Triton, so models that fluently write GPU kernels routinely hallucinate Pallas APIs, emit memory-space annotations that do not type-check, or violate systolic tiling constraints in ways that no amount of generic compile feedback can resolve. GPU benchmarks therefore cannot be directly applied. Workloads, problem sizes, programming abstractions, and optimization strategies are all TPU-specific.

[*]Equal contribution. Work done while at Google. [1]Harvard University [2]Google [3]UC Berkeley [4]Google DeepMind. Correspondence to: Arya Tschand <aryatschand@g.harvard.edu>, Sethu Sankaran <sethusankaran@google.com>.

[1]Code: https://github.com/AI-Hypercomputer/accelerator-agents/tree/main/JAXBench

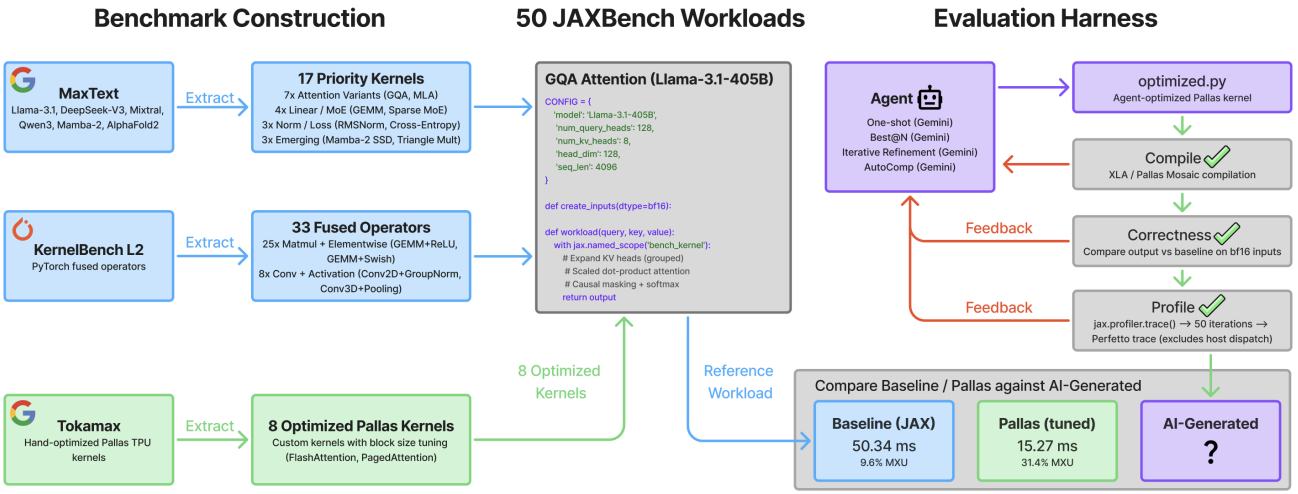


Figure 1. **JAXBENCH overview, a TPU-native benchmark for evaluating AI-generated Pallas kernels against hand-tuned baselines on TPU v6e.** 50 JAX reference workloads (17 priority kernels from MaxText LLMs + 33 fused operators from KernelBench L2) are sized to saturate TPU v6e MXUs. 8 priority kernels include Tokamax hand-tuned Pallas implementations as expert upper bounds. The agent harness evaluates LLM-driven methods (one-shot, best-of-N, iterative refinement with/without TPU documentation, and Autocomp). Generated kernels are compiled, correctness-checked on bf16 inputs, and profiled via `jax.profiler`.

Prior work with MultiKernelBench (Wen et al., 2025) extends benchmarking to Pallas but evaluates on TPU v2-8 (a legacy architecture), translates PyTorch workloads at problem sizes too small to saturate the MXU, and includes neither production JAX LLM operators nor optimized references that bound achievable performance. This sizing choice is consequential because at small shapes, workloads are dominated by memory traffic and launch overhead, so measured speedups do not reflect genuine algorithmic or scheduling improvement. A useful TPU benchmark must push workloads into the compute-bound regime where the MXU is the binding resource and tiling, pipelining, and layout choices affect throughput.

We introduce JAXBENCH, a benchmark suite of 50 JAX workloads for TPU kernel optimization on TPU v6e (Trillium). Our contributions are as follows.

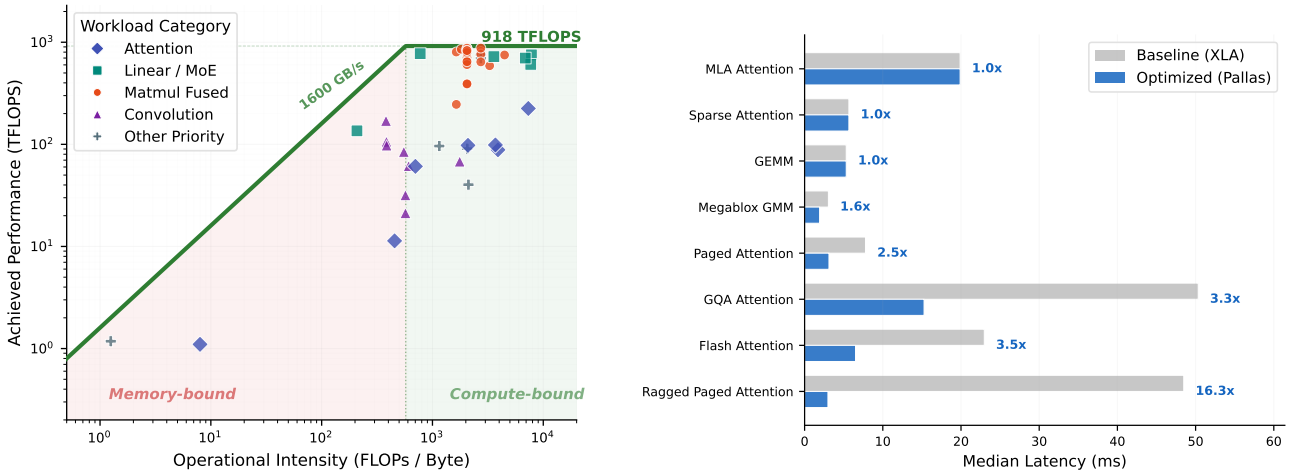
1. **TPU-native workload suite.** 17 production operators from real LLM architectures (Llama-3.1, DeepSeek-V3, Mixtral, Mamba-2, RetNet, AlphaFold2) extracted from MaxText (Google AI Hypercomputer, 2022), and 33 fused operator sequences from KernelBench Level 2 translated from PyTorch to JAX with problem sizes adjusted to achieve high TPU MXU utilization. For 8 priority kernels, we additionally provide hand-optimized Pallas TPU kernels from the upstream JAX Pallas ops library (Tokamax) with tuned block sizes.
2. **Rigorous evaluation harness.** Device-side profiling via `jax.profiler` eliminates dispatch overhead, yielding reproducible per-iteration kernel timings.
3. **Agent evaluation.** We evaluate one-shot LLMs, iterative refinement, and Autocomp (Hong et al., 2025) (augmented with TPU architecture documentation) on the benchmark and give insights into failure modes.

JAXBENCH addresses a key shortcoming in existing kernel generation benchmarks, and a full comparison to related work and summary of the unique design principles and contributions is given in Appendix A.

2. JAXBench

JAXBENCH (Figure 1) is built on three principles. First, *relevant workloads* from real LLM architectures. Second, *headroom to optimize* via problem sizes that push XLA into compute-bound regimes so optimization measures genuine improvements rather than launch overhead. Third, *reproducible baselines* via idiomatic JAX with `jax.jit` (XLA performs aggressive fusion and layout optimization) with hand-tuned Pallas references where available.

Priority kernels (17 workloads). We extract 17 compute-critical operators from production LLMs in MaxText (Google AI Hypercomputer, 2022), spanning attention variants (flash (Dao et al., 2022), GQA (Grattafiori et al., 2024), MLA (Liu et al., 2024), sparse/splash (Jiang et al., 2024), flex, paged, and ragged paged (Jiang et al., 2026)), dense and sparse linear algebra (GEMM, SwiGLU MLP, sparse MoE, Megablox GMM, ragged dot), normalization and loss, and emerging architectures (RetNet, Mamba-2 SSD, AlphaFold2 triangle multiplication). Each uses production-scale dimensions. The GQA workload, for example, uses Llama-3.1-405B dimensions with 128 query heads and 8 KV heads at sequence length 4096. For 8 of these we additionally import equivalent Pallas kernels from



(a) TPU v6e roofline with all 50 workloads.

(b) Tokamax Pallas speedup over XLA.

Figure 2. **JAXBENCH workloads span the full TPU roofline, and Pallas yields up to a 16× speedup.** Matmul-fused and linear/MoE kernels reach 60–95% MXU utilization under XLA, while some attention and elementwise kernels remain memory-bound. Hand-optimized Pallas closes the gap on the 8 priority workloads (up to 16.3× on Ragged Paged Attention, 3.5× on Flash Attention).

the upstream JAX Pallas ops library (Tokamax) and tune block sizes via exhaustive grid search on TPU v6e (203 configurations, yielding up to 2.79× speedup over Pallas defaults on Megablox GMM). These optimal block sizes are TPU-version-specific and should be re-tuned on each target.

KernelBench L2 (33 workloads). We translate 33 KernelBench Level 2 fused operators (Ouyang et al., 2025) from PyTorch to JAX via Gemini-driven translation with `jnp.allclose` validation at bf16 tolerance (atol = rtol = 10^{-2}). We choose L2 because it instantiates the matmul/conv × activation/norm/pooling fusion structures most relevant for TPU optimization, deduplicate by fusion signature, and retain one per class. Each workload's free dimensions are swept and frozen at the smallest configuration where XLA reaches ≥ 60% MXU utilization, yielding compute-bound shapes (e.g. batch 4096 with 8192 × 8192 features in bf16 for matmul-fused operators). Every workload exposes a standard interface (`CONFIG`, `create_inputs(dtype)`, and a `workload(*inputs)` entry point annotated with `jax.named_scope`). Two workloads with data-dependent control flow run eagerly.

Timing. Accurate kernel timings that provide consistent feedback to an optimization agent require device-side measurement. Wall-clock `time.perf_counter()` with `block_until_ready()` includes Python/JAX dispatch overhead, which can be 10–20% of measured time for kernels under 1 ms. We instead use `jax.profiler.trace()` Perfetto traces, running 5 warmup + 50 timed iterations, extracting `jit_*` device events, and reporting the median. FLOPs come from XLA's `cost_analysis()` on compiled HLO. The FLOP/byte derivation for ragged paged attention is in Appendix B.

Table 1. Cross-method comparison on the full 50-benchmark JAXBench suite with Gemini 3 Flash. Geomean is per-benchmark best speedup floored at 1×. Correctness is the number of benchmarks each method solved at least once.

Method	Geomean	Correctness
Best-of-N	1.01×	13/50
Iterative refinement	1.18×	32/50
Iterative + context	1.28×	**48/50**
Autocomp	**1.36**×	45/50

3. Evaluation

We evaluate generated kernels on (1) compilability to a valid JAX/Pallas program, (2) numerical correctness on bf16 inputs against the reference, and (3) wall-clock speedup over the XLA baseline (our headline metric, with per-benchmark best-so-far floored at 1× before geomean aggregation). We also report **fast**$_1$@N, the fraction of benchmarks whose best-so-far sample beats XLA after N cumulative samples.

Methods. We evaluate four classes of methods. **Best-of-N.** Independent one-shot completions conditioned on a short TPU-v6e preamble and the JAX source. **Iterative refinement.** 18 chains × 8 turns, each turn appending a feedback block (correctness verdict, latency/speedup, stdout, profiler summary). **Iterative + context.** Same setup with Autocomp's TPU-specific context (see Appendix E.2) prepended, isolating context from search structure. **Autocomp** (Hong et al., 2025). Beam search (4-iter translation +

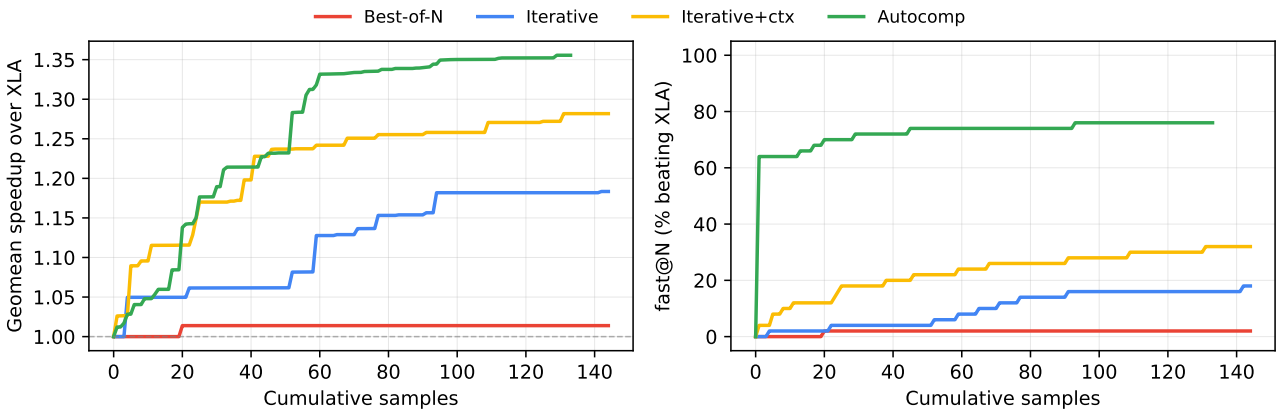


Figure 3. **Sample-efficiency trajectories on the 50-benchmark JAXBench suite with Gemini 3 Flash,** including best-so-far geomean speedup (left) and $\text{fast}_1@N$ (right) as a function of cumulative samples.

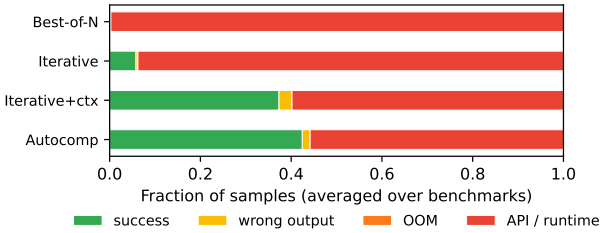


Figure 4. **Per-sample outcome breakdown on JAXBench (Gemini 3 Flash).** Injecting TPU documentation raises per-sample correctness from 5.8% (iterative) to 37.3% (iter+ctx) without changing the search algorithm. "Wrong output" is samples that run but fall outside numerical tolerance, and "API / runtime" covers Pallas API misuse and compile/runtime exceptions, dominating context-free failures (99.7% of best-of-N, 93.8% of iterative).

4-iter optimization, beam size 3, 6 candidates generated per beam element) over the same documentation. All four use a 144-sample budget per benchmark, with full prompts in Appendix E. We run Gemini 3 Flash (Team et al., 2023) on all 50 benchmarks plus a 5-kernel Gemini 3.1 Pro ablation.

Main results. Table 1 and Figure 3 summarize the full 50-benchmark Flash evaluation. The four methods separate cleanly into a hierarchy where each added layer of agent structure (feedback, context, beam search) produces a measurable jump in correctness and geomean speedup. Best-of-N beats XLA on only 1 benchmark. Plain iterative refinement improves modestly to 1.18×. Adding TPU-specific documentation sharply reduces API misuse; iterative+context raises per-sample correctness from 5.8% to 37.3% without changing the search algorithm, lifting geomean to 1.28× and solving 48/50 benchmarks. Autocomp's structured beam search converts its 45/50 correct kernels into a 1.36× geomean with 76% beating XLA. The two top methods illustrate a clear breadth/depth tradeoff. Iterative+context debugs longer and solves *more* benchmarks (48 vs. 45), whereas Autocomp does not debug and focuses solely on modifying known correct code (throwing out any failing implementations), landing a higher geomean on the kernels for which it does generate correct code. However, API misuse remains the most common outcome even with context (Figure 4), indicating opportunities for future work.

Hand-tuned Pallas comparison. On the 8 priority operators with Tokamax references, Tokamax achieves a 2.08× geomean (also floored at 1× per benchmark) over XLA. Autocomp reaches 1.60× (≈ 77% of hand-tuned), exceeding Tokamax on 2 kernels while failing altogether to generate correct code for paged and ragged paged attention. Iterative+context reaches 1.34×. The full breakdown can be found in Appendix C.

Model-capacity ablation. On a 5-kernel subset, Autocomp with Gemini 3.1 Pro reaches 49.1% per-sample correctness and 3.13× geomean, above its Flash counterpart. This gain is smaller than the Flash-only gain from injecting Autocomp's documentation into iterative refinement. Within the model-capacity range we tested, target-specific context is a stronger lever than raw model scale (Appendix D).

4. Discussion

JAXBENCH currently runs only on a single TPU v6e chip. The most important extension is multi-chip support, since real workloads rely on sharding and collective primitives like `shard_map`, `psum_scatter`, and `all_to_all`, which appear in pretraining data even less often than Pallas. Hand-tuned references also cover only 8 of the 17 priority operators. Two next steps follow naturally from our results. The first is using JAXBENCH's correctness and latency signals as RL or fine-tuning rewards on Pallas. The second is building hybrid controllers that route between iterative refinement and beam search per benchmark.

References

Cao, S., Mao, Z., Gonzalez, J. E., and Stoica, I. K-search: Llm kernel generation via co-evolving intrinsic world model. *arXiv preprint arXiv:2602.19128*, 2026.

Dao, T., Fu, D., Ermon, S., Rudra, A., and Ré, C. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022.

Google AI Hypercomputer. MaxText: A high performance, scalable, open-source LLM in pure Python/JAX. https://github.com/AI-Hypercomputer/maxtext, 2022. Accessed: 2026-04-26.

Grattafiori, A., Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Vaughan, A., et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Hong, C., Bhatia, S., Cheung, A., and Shao, S. Autocomp: Llm-driven code optimization for tensor accelerators. In *Machine Learning for Computer Architecture and Systems 2025*, 2025.

JAX Development Team. Writing TPU kernels with pallas. https://docs.jax.dev/en/latest/pallas/tpu/details.html, 2024. Accessed: 2026-04-26.

Jiang, A. Q., Sablayrolles, A., Roux, A., Mensch, A., Savary, B., Bamford, C., Chaplot, D. S., Casas, D. d. l., Hanna, E. B., Bressand, F., et al. Mixtral of experts. *arXiv preprint arXiv:2401.04088*, 2024.

Jiang, J., Chen, Y., Hechtman, B. A., Zhang, F., and Mu, Y. Ragged paged attention: A high-performance and flexible llm inference kernel for tpu. *arXiv preprint arXiv:2604.15464*, 2026.

Lange, R. T., Sun, Q., Prasad, A., Faldor, M., Tang, Y., and Ha, D. Towards robust agentic cuda kernel benchmarking, verification, and optimization. *arXiv preprint arXiv:2509.14279*, 2025.

Li, J., Li, S., Gao, Z., Shi, Q., Li, Y., Wang, Z., Huang, J., WangHaojie, W., Wang, J., Han, X., et al. Tritonbench: Benchmarking large language model capabilities for generating triton operators. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 23053–23066, 2025.

Liao, G., Qin, H., Wang, Y., Golden, A., Kuchnik, M., Yetim, Y., Ang, J. J., Fu, C., He, Y., Hsia, S., et al. Kernelevolve: Scaling agentic kernel coding for heterogeneous ai accelerators at meta. *arXiv preprint arXiv:2512.23236*, 2025.

Liu, A., Feng, B., Xue, B., Wang, B., Wu, B., Lu, C., Zhao, C., Deng, C., Zhang, C., Ruan, C., et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

Ouyang, A., Guo, S., Arora, S., Zhang, A. L., Hu, W., Ré, C., and Mirhoseini, A. Kernelbench: Can llms write efficient gpu kernels? *arXiv preprint arXiv:2502.10517*, 2025.

Team, G., Anil, R., Borgeaud, S., Alayrac, J.-B., Yu, J., Soricut, R., Schalkwyk, J., Dai, A. M., Hauth, A., Millican, K., et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

Thakkar, V., Ramani, P., Cecka, C., Shivam, A., Lu, H., Yan, E., Kosaian, J., Hoemmen, M., Wu, H., Kerr, A., Nicely, M., Merrill, D., Blasig, D., Atluri, A., Qiao, F., Majcher, P., Springer, P., Hohnerbach, M., Wang, J., and Gupta, M. CUTLASS. https://github.com/NVIDIA/cutlass, 2023. Accessed: 2026-04-26.

Tillet, P., Kung, H.-T., and Cox, D. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pp. 10–19, 2019.

Tschand, A., Awad, M., Swann, R., Ramakrishnan, K., Ma, J., Lowery, K., Dasika, G., and Reddi, V. J. Swizzleperf: Hardware-aware llms for gpu kernel performance optimization. *arXiv preprint arXiv:2508.20258*, 2025.

Wen, Z., Zhang, Y., Li, Z., Liu, Z., Xie, L., and Zhang, T. Multikernelbench: A multi-platform benchmark for kernel generation. *arXiv e-prints, pp. arXiv–2507*, 2025.

Xing, S., Zhai, Y., Jiang, A., Dong, Y., Wu, Y., Ye, Z., Ruan, C., Huang, Y., Zhang, Y., Yin, L., et al. Flashinfer-bench: Building the virtuous cycle for ai-driven llm systems. *arXiv preprint arXiv:2601.00227*, 2026.

A. Related Work and Benchmark Comparison

Table 2 positions JAXBENCH against existing LLM kernel-generation benchmarks. JAXBENCH is the only suite targeting contemporary TPUs with production-scale workloads, hardware-saturating problem sizes, and expert-written Pallas reference kernels for a subset.

Table 2. Comparison of LLM kernel-generation benchmarks.

Benchmark	Target	# Tasks	Workload Source	Expert Refs	Sat. Sizes
KernelBench (Ouyang et al., 2025)	CUDA / GPU	250	PyTorch modules	—	Yes
TritonBench (Li et al., 2025)	Triton / GPU	184	Real repos	Triton	Partial
FlashInfer-Bench (Xing et al., 2026)	CUDA / GPU	–	LLM serving traces	FlashInfer	Yes
MultiKernelBench (Wen et al., 2025)	Pallas / TPU v2-8	285	PyTorch modules	—	No
JAXBENCH (ours)	**Pallas / TPU v6e**	**50**	**Production LLMs + Standard JAX**	**Pallas**	**Yes**

B. FLOP/Byte Derivation for Ragged Paged Attention

We compute achieved MXU utilization on TPU v6e and operational intensity of ragged paged attention (Jiang et al., 2026) as

$$\text{MXU util}_{\text{v6e}} = \frac{\text{FLOPs}/t_{\text{median}}}{918 \times 10^{12}}, \qquad \text{OI}_{\text{ragged-attn}} = \frac{\text{FLOPs}_{\text{v6e}}}{\text{MemBW}_{\text{v6e}}} = \frac{G \sum_{i=1}^{B} L_i}{GB + \sum_{i=1}^{B} L_i} \tag{1}$$

where 918 TFLOPS is the TPU v6e single-chip bf16 peak (2× MXU, each a 256×256 systolic array at 459 TFLOPS), $G = H_q/H_{kv}$ is the GQA grouping factor, B is the number of sequences, and $\{L_i\}$ are the per-sequence KV lengths. The numerator counts attention FLOPs ($H_q \sum_i L_i$ since every query head attends over its full KV sequence). The denominator counts bytes read ($H_q B$ from per-sequence queries, $H_{kv} \sum_i L_i$ from the shared KV cache). Dividing both by H_{kv} yields the closed form above. In the long-context limit $\sum_i L_i \gg GB$ the operational intensity approaches G, reflecting that each KV token is reused by exactly G query heads. For Llama-3.1-70B ($G = 8$), this upper bound sits far below the v6e ridge point ($\text{OI} = 574$), confirming that ragged paged attention is memory-bandwidth-bound at any context length.

C. Hand-Tuned Pallas Per-Kernel Breakdown

Table 3. Speedups on the 8 priority kernels with hand-tuned Pallas references. Agents use a 144-sample Gemini 3 Flash budget. "—" denotes no correct kernel. Bold marks the best method per row.

Benchmark	XLA (ms)	HT (ms)	Hand-tuned	Autocomp	Iter+ctx	Iter
Flash Attention	25.21	6.45	**3.91**×	2.67×	0.46×	—
GQA Attention	51.13	14.59	**3.50**×	2.63×	2.63×	—
MLA Attention	20.32	21.72	0.94×	0.85×	**1.54**×	—
Sparse Attention	5.95	6.89	0.86×	**2.81**×	2.54×	—
Paged Attention	7.82	3.41	**2.29**×	—	0.61×	—
Ragged Paged Attention	18.70	2.71	**6.91**×	—	—	—
GEMM	5.36	5.59	0.96×	0.75×	**0.97**×	0.59×
Megablox GMM	3.26	2.01	1.62×	**2.21**×	0.27×	0.52×
Geomean (floor 1×)			**2.08**×	1.60×	1.34×	1.00×

D. Gemini 3.1 Pro Ablation

To measure the effect of model capacity, we rerun best-of-N, iterative refinement, and Autocomp on a 5-kernel subset using Gemini 3.1 Pro and compare against the same three methods run on the same 5 benchmarks with Gemini 3 Flash, so that Flash and Pro are compared on identical benchmarks and sample budgets. On this subset, Autocomp with Gemini 3.1 Pro reaches 49.1% per-sample correctness and a $3.13\times$ geomean speedup over XLA, both above its Flash counterpart. The Pro-over-Flash gain is smaller than the Flash-only gain from injecting Autocomp's documentation into iterative refinement

on the full 50-benchmark suite, indicating that target-specific context is a stronger lever on Pallas correctness than raw model scale within the capacity range we evaluate.

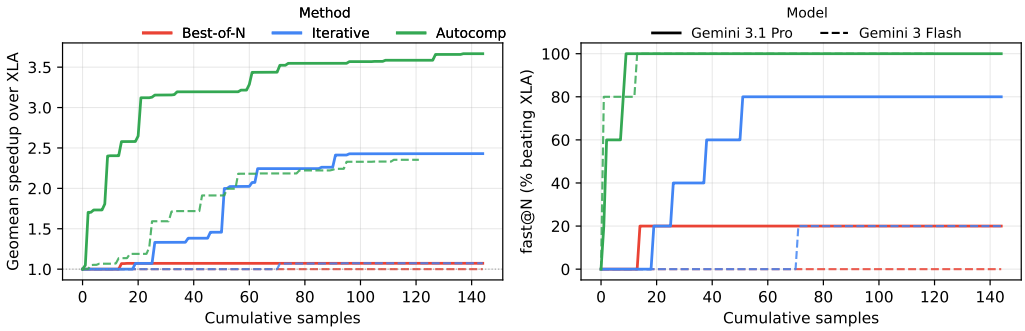


Figure 5. Model-capacity ablation on the 5-kernel subset (RMSNorm, Flex Attention, RetNet Retention, Mamba-2 SSD, Flash Attention) showing speedup over XLA (left) and $\text{fast}_1@N$ (right) vs. cumulative samples. Solid lines are Gemini 3.1 Pro and dashed lines are Gemini 3 Flash measured on the same 5-kernel subset (distinct from the full-50 Flash trajectories in Figure 3).

E. Baseline Prompts and Agent Context

This appendix documents the exact prompt construction for each evaluated method so the setup can be reproduced from the paper text alone.

E.1. Best-of-N and Iterative Refinement

Both baselines use the same short system preamble:

```
You are optimizing a JAX kernel for TPU v6e (Trillium) using the Pallas
programming model (jax.experimental.pallas).
Target hardware:  TPU v6e.  VMEM = 128 MiB, 8 TensorCores per chip, peak ≈918
TFLOPS bf16, HBM ≈1526 GB/s.
Rules for every turn:  (1) keep the public workload(*inputs) signature unchanged;
(2) output a single complete Python file inside one ```python...``` block; (3) do
not include explanatory prose outside the code block.
Strategy:  if you do not yet have a correct Pallas implementation, your first
priority is to produce a correct, straightforward translation, even if it isn't
faster than the XLA baseline.  Only after you have a correct Pallas kernel should
you focus on optimizing it for speed.
```

Best-of-N. Each of the N samples is an independent completion of a prompt consisting of the preamble above, a single line `Benchmark: <prob_id>`, a one-sentence task description ("Below is the XLA/JAX reference implementation. Rewrite it as a Pallas kernel that produces identical outputs and runs faster."), and the JAX source wrapped in a Python code block. No feedback is provided.

Iterative refinement. The first turn of each chain uses the same prompt as Best-of-N. Each subsequent turn within a chain appends the following feedback block, then asks the model for a new complete Pallas implementation:

- If the previous attempt was correct: "Previous attempt was correct. Latency: <x> ms (XLA baseline: <y> ms, speedup: <s> ×). Best-so-far in this chain: <z> ms."
- If the previous attempt was incorrect: "Previous attempt was incorrect or failed to run." followed by the last 40 lines of the runner's stdout.
- A profiler summary (top operators by time) from the previous run when available.
- The previous implementation's full source, wrapped in a Python code block.

Chains run independently and never share context, and 18 chains of 8 turns give a total of 144 samples per benchmark.

Iterative refinement with Autocomp context. The *iterative+context* variant uses the same chain/turn/feedback layout as plain iterative refinement (18 chains $\times$ 8 turns, same feedback block on correctness or failure, same final-code echo) with one change. Before every prompt we prepend the static portion of Autocomp's agent context, namely the hardware architecture summary, the per-benchmark-selected Pallas API reference, the selected code examples, and the rules block (see next subsection). The per-benchmark category and example selections are made by the same LLM yes/no filter Autocomp uses and are cached per benchmark, so a chain's eight turns see the same context menu. The variant does not use Autocomp's beam search or translate/optimize phase split, and is purely a context injection into the iterative baseline.

E.2. Autocomp Context Artifacts

We passed four JAX and Cloud TPU documentation sources into Autocomp's Agent Builder pipeline:

- JAX Pallas guide: `https://docs.jax.dev/en/latest/pallas/index.html`
- JAX Pallas TPU guide: `https://docs.jax.dev/en/latest/pallas/tpu/index.html`
- `jax.experimental.pallas.tpu` API reference: `https://docs.jax.dev/en/latest/jax.experimental.pallas.tpu.html`
- Cloud TPU documentation: `https://docs.cloud.google.com/tpu/docs/`

Each source is crawled with `max_depth=2` and `max_pages=250`. An LLM-driven routing-and-synthesis pass then distills the crawled content into four textual artifacts that together form the static portion of every agent prompt:

- **Hardware architecture summary** ($\approx$8 KB): a single-pass LLM summary of TPU v6e characteristics (memory hierarchy, compute units, bandwidth and capacity numbers, and programming-model constraints).
- **Pallas API / ISA reference** ($\approx$115 KB): per-API entries (signature, description, parameter semantics, inline usage snippets) extracted verbatim from the source documentation and grouped into functional categories. At prompt time, an LLM yes/no filter selects which categories are relevant to the current kernel and caches the selection per-benchmark.
- **Annotated code examples** ($\approx$30 KB): representative Pallas kernels drawn from the ingested documentation, each accompanied by a one-line summary. Examples are selected per-benchmark via a parallel LLM yes/no filter.
- **Rules** ($\approx$6 KB): correctness constraints (e.g., block-shape divisibility, memory-space annotations, aliasing rules) extracted as imperative bullet points and appended to every planning and coding prompt.

During beam search, each candidate-generation prompt concatenates the architecture summary, the per-benchmark-selected API reference, any selected code examples, the candidate's current code (plus its latest score and any hardware-counter feedback), and the rules block, before asking the model for either a plan or a complete optimized kernel. The translate and optimize phases use the same scaffold but draw from separate menus of strategies.

F. Limitations and Future Work

All 50 workloads run on a single TPU v6e chip, and multi-chip sharding and collective communication are out of scope for this release. Hand-tuned Pallas references with grid-searched block sizes are available for 8 of the 17 priority operators and serve as an aspirational upper bound. The Gemini 3.1 Pro evaluation is restricted to a 5-kernel subset as a controlled model-capacity ablation rather than a primary headline result.

The most consequential extension is the move to multi-TPU evaluation, where end-to-end performance is governed not only by per-chip kernel quality but also by sharding, collective scheduling, and overlap of communication with compute. A natural successor benchmark would extend JAXBENCH along the parallelism axis to tensor-parallel, pipeline-parallel, and expert-parallel sharding on multi-chip pods. Such an extension would test whether the documentation-versus-search separation we observe at the single-chip level generalizes to collective scheduling, where the relevant primitives (`shard_map`, `psum_scatter`, and `all_to_all`, plus asynchronous collectives and overlapping pipelines) are even more sparsely represented in pretraining data than Pallas itself. Beyond multi-chip, the information-gap result motivates fine-tuning or RL

from execution feedback on curated Pallas corpora, for which JAXBENCH correctness and latency signals are directly usable as rewards. The tier structure observed in our per-kernel breakdown also suggests hybrid controllers that route between iterative refinement and beam-search agents on a per-benchmark basis.

G. Per-Benchmark Results

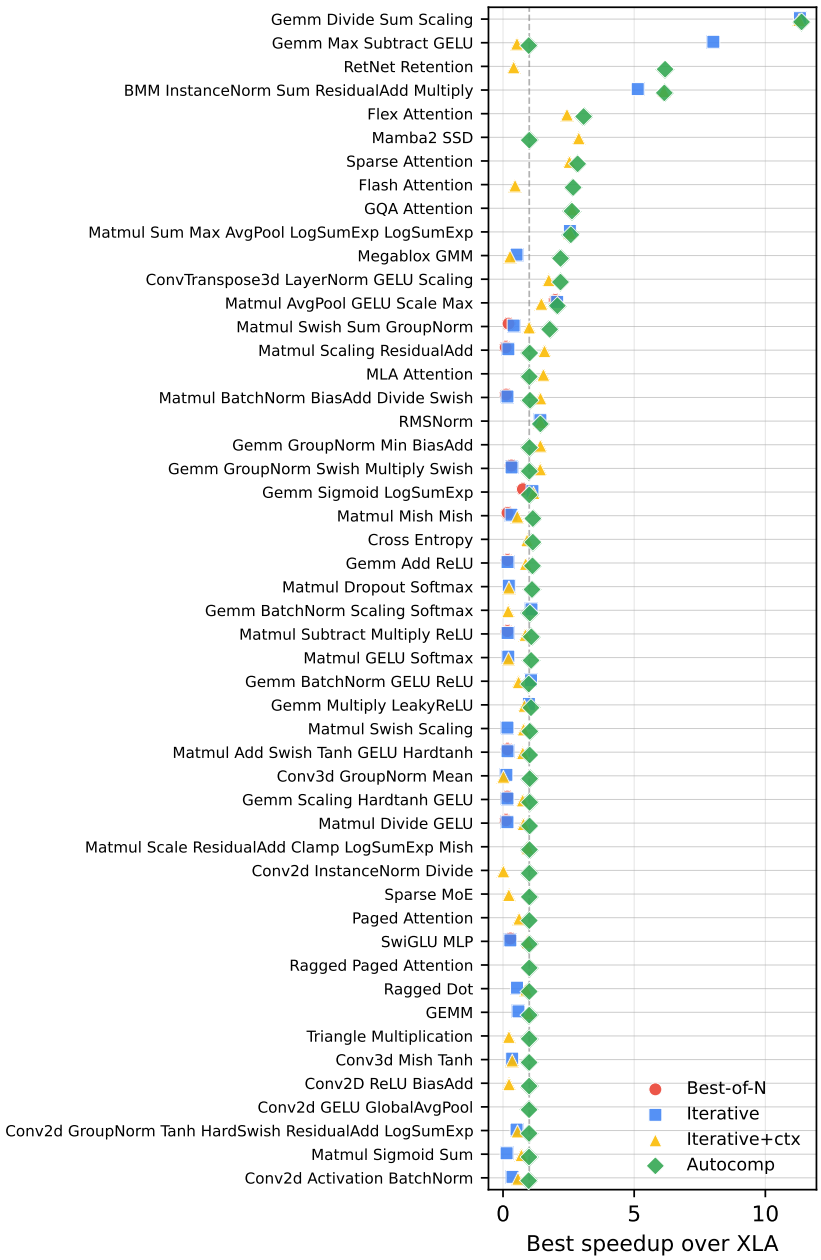


Figure 6. Best speedup over XLA for each benchmark and method, Gemini 3 Flash. Benchmarks sorted by best speedup achieved by any method. Missing markers indicate no correct implementation within the sample budget.