TPU Inference Externalization Full Steam Ahead - InferenceX

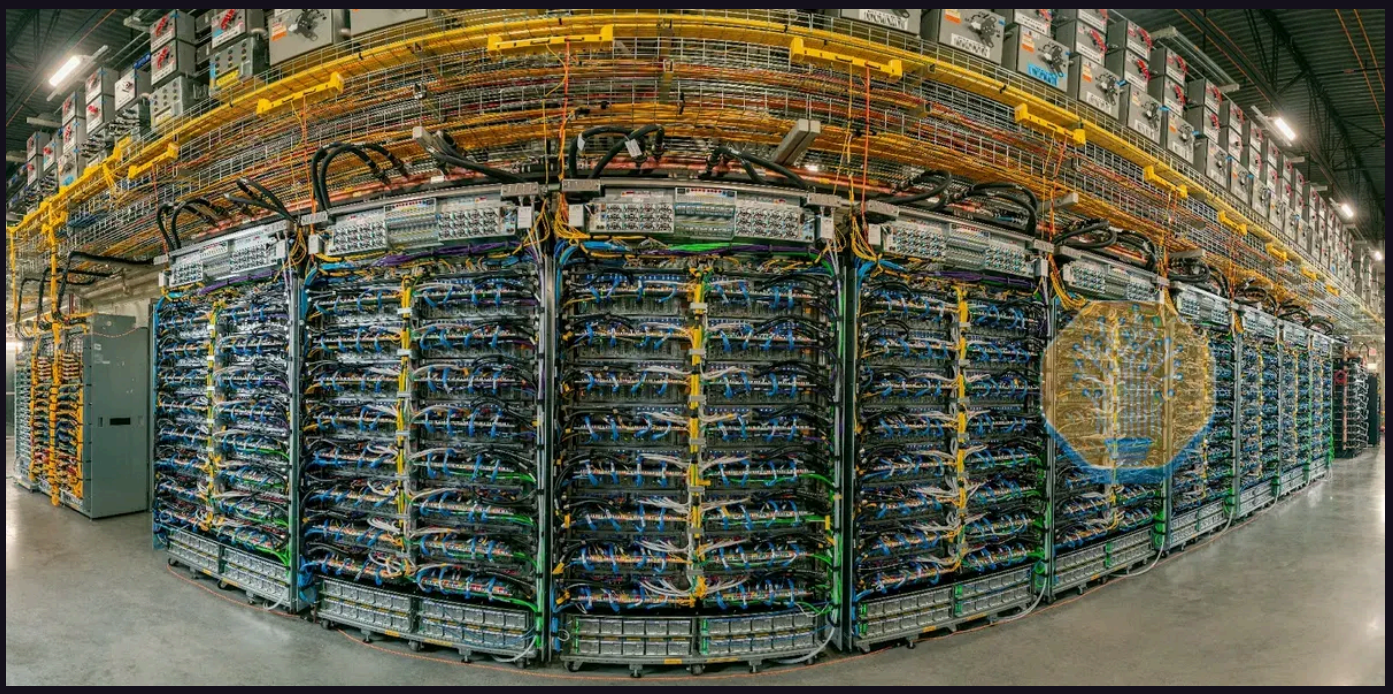
InferenceX, Up to 50% Better Performance per Dollar, Rapid Externalization of TPU stack, Growing Customer Base, Ironwood, TPUv8i, Reducing CUDA Moat

ALEC IBARRA, CAM QUILICI, BRYAN SHAN, AND 4 OTHERS

SEP 08, 2026 • PAID

57 4 Share

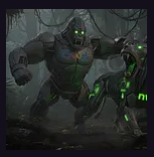
For more than a decade, the industry has watched Google build an empire on its own silicon. Search, Ads, YouTube, and every generation of Gemini run on TPUs. Few accelerators have attracted as much architectural scrutiny or as much debate about what their performance and economics would look like outside the company that designed them. Anthropic being the biggest user of TPUs, surpassing Deepmind's own use by 2029.



Source: Google

Google's internal success was never the question. The question was how much of that advantage the rest of the industry could actually get. Could you take an open-weight model, serve it through a familiar inference engine, and beat NVIDIA on the economics that matter to your business?

Ironwood delivers up to 50% better performance per dollar. Its advantage extends across much of the Pareto curve, and we examine the economics from both sides: Google's internal total cost of ownership and the external TCO an actual customer pays.



TPUv7: Google Takes a Swing at the King

DYLAN PATEL, MYRON XIE, AND 10 OTHERS · 2025年11月28日

Read full story →

Ironwood (TPUv7) is the first generation in which Google is competing for others' inference workloads with chips that can be purchased outright or rented through its own cloud. In November 2025, we already said that Anthropic loves TPUs and committed to over one million of them (around 400k+ in direct purchases and 600k+ rented through GCP), used mainly for training but also for inference. Our Accelerator Model has the latest figures for Anthropic's TPU shipments and Google's overall TPU shipments by quarter, plus estimates for TPUv8i, v8t, and various v9 / v10, and more

We are excited by how quickly the new TorchTPU stack is developing, the external stack for TPUs. Later in the article, we will discuss the upcoming work needed for TPU software externalization, including optimizing speculative decoding, disaggregated prefill, KV-cache offloading, multi-turn agentic workloads, and more. Even so, we at SemiAnalysis **strongly believe that TPU externalization is heading in the right direction and moving full steam ahead.** Furthermore, unlike AMD, which is still learning how to build a test-first software culture, Google has decades of software engineering experience and an extremely well established quality-driven culture, so we expect external TPU software to mature rapidly.

In this article, we will cover all the optimizations that went into TPU kernels and the serving stack for open-weight models, including DP attention optimization, MoE routing and kernel optimization, and reducing padding in GDN kernels. We will also take a deep dive into the TPU system and discuss the next steps the amazing TPU performance engineers are pursuing to make the stack widely available.

Google has spent more than a decade demonstrating what it can build with TPUs. Now we get to measure what the rest of the industry can do with them.

Shoutout to the Google (Chris Chan, Jahangir Hasan, Wangyuan Zhang, Anne Stern, Puneith Kaul, Ruizi Dong, Sangam Jindal, Qi Zhou, Madhan Jaganathan, Gang Ji, Jun Wan, Devanshu Jain, Jiaxin Cao, Srinath Mandalapu, Haowen Ning) and Inferact teams for this

InferenceX Official Preview: TPUv7 Ironwood vs. Blackwell and Blackwell Ultra

We are already seeing strong results from the upcoming native TorchTPU vLLM stack in apples-to-apples comparisons against Nvidia GPUs. Google is using Qwen3.5 397B in FP8 as the initial bring-up model. Later sections take a deep dive into why the new TorchTPU approach is a marked improvement over the previous TorchAX path for external TPU vLLM/SGLang serving.

Type your email... Subscribe

Once that foundation is in place, Google plans to extend support to other open-weight models, including Kimi K3 and GLM5.3. Once a handful of models are well optimized, we believe adding optimized support for a wide range of popular open models near day 0 becomes far easier. Today, vLLM and SGLang concentrate their day-0 support on Nvidia, with passable day-0 coverage for AMD. We expect TorchTPU to be stable enough in the near future that vLLM and SGLang maintainers may add TPUs to that day-0 list. The stack is expected to leave private beta & be open sourced around October.

In apples-to-apples comparisons of aggregated serving with FP8 and single-token prediction, we are seeing up to 50% better performance per dollar from TPU than from B200 and B300 running FP8 in aggregated serving. When serving models using FP4 on NVIDIA GPUs, there is quality loss verus FP8. TPUv7 does not have native FP4 computation thus on FP4, NVIDIA GPUs still maintains the lead. This will change with TPUv8i which has native FP4 support thus we strongly believe TPUv8i Boardfly will be competitive to Rubin NVL72.

We believe the TPU performance-per-dollar advantage can persist once Google enables disaggregated serving and speculative decoding, with MTP and disaggregated serving enabled on both sides of the comparison. Later in the article, we discuss the optimizations Google plans to add once the TorchTPU foundation is in place. To keep the bring-up scope contained, Google has initially focused on benchmarking 8k1k. However, we believe that the TorchTPU stack will also perform amazingly well on agentic workloads & will be

TPU is King on Performance per Dollar

At an interactivity of 100 tokens/s/user, Ironwood costs approximately $0.181 per million total tokens, compared with $0.222 for B200 and $0.276 for B300. That is approximately 19% lower cost than B200 and 34% lower than B300, while delivering the same per-user generation speed.

For this comparison, we use an external TPU TCO against B200/B300 TCO for an hyperscaler lab to buy TPU. Last year, SemiAnalysis Accelerator Model was the first to break that Google is selling TPUs instead of just renting out through Google Cloud. TPU's lower hourly cost offsets its lower throughput across much of the interactivity range. The SemiAnalysis TCO Model provides a full breakdown of the TPU BoM estimate and TCO estimates.

Source: SemiAnalysis InferenceX Preview

Source: InferenceX Preview

Across most of the raw-performance curve, TPU does not outperform Nvidia GPUs. However, that comparison does not account for TPU's lower TCO. **Buyers of TPU are mainly concerned with how much revenue they can make, i.e. performance per dollar and performance per watt.**

That being said, at interactivity of 20 tok/s/user, Ironwood also leads on raw throughput, reaching 9,364 total tokens/s/chip, compared with 8,903 on B200 and 8,925 on B300. This is approximately 5% higher throughput than either GPU in these runs. Combined with the lower modeled hourly cost, Ironwood delivers 50.4% more tokens per dollar than B200 and 96.0% more than B300.

Source: SemiAnalysis InferenceX Preview

If we consider TPU TCO for Google's internal workloads at $1.03/chip-hour, the performance-per-dollar advantage at concurrency 256 increases to 76.7% over B200 and 130.2% over B300. However, these high-concurrency results come with latency trade-offs. For example, at concurrency 256, TPU mean TTFT is 5.41 seconds, compared with 3.75 seconds on B200 and 2.40 seconds on B300. The previously discussed 50% to 96% advantage applies to this datapoint specifically, rather than every latency target.

Source: SemiAnalysis InferenceX Preview

Looking at end-to-end latency, Ironwood remains competitive beyond that highest-throughput point. At a 20-second median response time, the Pareto curves put TPU at approximately $0.098 per million total tokens, compared with $0.106 for B200 and $0.132 for B300. TPU therefore costs approximately 8% less than B200 and 25% less than B300.

Source: SemiAnalysis InferenceX Preview

Despite all of the above, B200 can still come out ahead on a small part of the Apples to Apples curve, including around the 30-second median response time. However, TPU regains its lower cost per million tokens at longer response times, while maintaining a cost advantage over B300 across the overlapping range shown here.

SemiAnalysis is a reader-supported publication.
To receive new posts and support our work,
consider becoming a subscriber.

Type your email... Subscribe

Apples to Bananas: GB300 NVL72 Disagg verus TPUv7 Agg

Google has been running disaggregated serving internally for years in production and that path is heavily optimized, but the external TPU serving stack does not have a fully optimized disagg path yet. So today, GB200/300 NVL72 is more competitive on perf per dollar in a disagg-vs-disagg comparison. But we expect that gap to close within a few

GB200/300 NVL72 disagg in a follow-up article. TPUv7 pods can scale up to over 1k+ chips through their low latency ICI fabric thus can enable optimizations for large models disagg & ultra wideEP that NVIDIA NVL72 can't perform.

Even on an apples to bananas comparison between TPUv7 aggregated serving verus GB300 NVL72 disagg, using internal TCO, TPUv7 is decently competitive on high latency & low to left medium latency. But when comparing on the right medium e2e latency, GB300 NVL72 disagg has an single advantage against TPuv7 aggregrated serving.

Source: SemiAnalysis

When looking at it GB300 NVL72 disagg verus TPUv7 agg, it is competitive at low and high e2e latency but in the middle, there is about an 30% perf per dollar advantage of GB300 against TPUv7 agg. We strongly believe once disaggregated serving on TPUv7 is optimized, it would be competitive across the full pareto against GB300 NVL72.

Source: SemiAnalysis

Inference Serving with the New Native, First-Class TorchTPU Backend vs. the Previous TorchAX Stack

Previous TorchAX vLLM Backend

Google previously used the TorchAX backend to translate the PyTorch-based implementations in vLLM and SGLang into JAX. TorchAX was intended to let developers run PyTorch model definitions through JAX. However, issues with this approach have prompted Google and the PyTorch, vLLM, and SGLang communities to collaborate on a new approach called TorchTPU. It lets developers use TPUs as native PyTorch devices, while StableHLO, XLA, and TPU-optimized Pallas kernels handle the low-level execution behind the scenes. In the upcoming sections of the article, we will be showing the performance of this new TorchTPU backend compared to NVIDIA Blackwell & Blackwell Ultra GPUs.

vLLM TPU support has evolved through three different stages:

- Its first TPU prototype used PyTorch/XLA. Its lazy execution model collected operations into computation graphs for XLA to compile, rather than executing each PyTorch operation immediately.

PyTorch model into operations that JAX can execute. The goal is to reuse mature TPU primitives and parallelism support without requiring every PyTorch model to be rewritten.

- The upcoming third design, TorchTPU, lets vLLM treat TPUs as native PyTorch devices. It will be the go-to backend for SGLang and vLLM on TPUs going forward.

The diagram below zooms in on the second stage and shows how vLLM's tpu-inference backend combines direct JAX models and PyTorch models running through TorchAX.

Source: SemiAnalysis/vLLM

The original choice of JAX was pragmatic. In the last announcement, Google described difficulties with low-level optimization, paged attention, and fitting vLLM's worker model to TPU execution. They thought that JAX would offer more mature TPU primitives and parallelism support. TorchAX let the team use those capabilities without rewriting every

With TorchAX, developers still write the model in PyTorch, but JAX executes it on the TPU. TorchAX acts as the translation layer between the two frameworks. As the TorchAX documentation explains, a torchax.tensor.Tensor behaves like an ordinary PyTorch tensor but is backed by a jax.Array. When the model invokes a tensor operation, known in PyTorch as an ATen operation, TorchAX intercepts it through the __torch_dispatch__ hook and translates it into one or more JAX operations. The model can therefore remain written in PyTorch even though JAX performs the computation.

vLLM must also manage state that changes during generation, especially the KV cache, which stores attention information from earlier tokens for reuse. Its model wrapper presents the model weights and KV cache to JAX as an explicit state. This process, known as functionalization, turns changes that would otherwise happen inside Python objects into inputs and outputs that the compiler can track. jax.jit can then capture each inference step as a computation graph and pass it through JAX's compilation pipeline for repeated execution.

From there, each component has a distinct role. Pallas supplies optimized TPU kernels for performance-critical operations. StableHLO represents the program in a standardized format that the compiler can process, and XLA converts it into executable code for the TPU. In vLLM's standard PyTorch path, torch.compile coordinates graph capture and compilation. In the TorchAX path, the JAX/XLA pipeline already performs those tasks, so vLLM bypasses its usual torch.compile route. These translation layers have caused many issues that the TorchTPU vLLM stack aims to solve.

Previous SGLang JAX Backend

Source: SemiAnalysis/SGLang

SGLang's current public TPU stack uses SGLang-JAX. SGLang-JAX makes a similar trade-off to vLLM's TorchAX backend but takes a different approach. It is a separate JAX-native serving engine rather than the PyTorch SGLang runtime translated through TorchAX. It combines SGLang-style scheduling and prefix caching with JAX models and TPU-specific kernels. Google and RadixArk have announced SGL-torchtpu as an additional PyTorch-native backend to extend the available serving paths.

Type your email... Subscribe

Shifting Toward the New Native, First-Class TorchTPU Backend

TorchTPU moves the framework boundary into PyTorch itself. It uses PyTorch's PrivateUse1 backend extension point to expose an ordinary torch.Tensor on device="tpu"

debugging, or call torch.compile for graph capture. In the compiled path, TorchDynamo and AOTAutograd produce an FX graph, TorchTPU lowers it to StableHLO, and XLA produces the TPU executable. Google's conference deck makes the compiler choice explicit that this path uses XLA, not Inductor and Triton.

Source: SemiAnalysis/PyTorch/Google

the kernel layer continues to use TPU-specific implementations. TorchTPU can call Pallas and JAX-backed custom kernels. Helion's TPU backend also emits Pallas. The user-facing framework becomes PyTorch-native, while the compiler and performance-critical kernels remain TPU-aware.

Developers can use .to("tpu") while retaining familiar distributed interfaces and serving-engine code. Google documents support for DDP, FSDP2, DTensor, one process per device, and both MPMD and SPMD execution. Those interfaces fit the process and distributed model already assumed by PyTorch serving engines.

For vLLM and SGLang, the opportunity is to reuse more upstream model code, schedulers, continuous batching, APIs, and feature logic instead of rebuilding them across a PyTorch-to-JAX boundary. That should reduce the cost of bringing up models and engine features, even when the final TPU implementation still needs specialized kernels.

Native PyTorch support does not eliminate the need for TPU-specific optimization. Pallas and TorchAX serve different purposes, TorchAX allows PyTorch model code to execute through JAX, while Pallas is used to implement performance critical operations directly for TPU hardware. Removing TorchAX doesn't mean removing the Pallas kernels underneath it. Since TorchTPU allows native Pytorch code to invoke Pallas and JAX kernels, kernels developed for earlier stack are able to migrate into the new stack.

Similar to GPU PyTorch vLLM, Custom kernels will still be needed for high performance inferencing. Tensor shapes and layouts still need to be tuned to use TPU's matrix multiplication units efficiently. We believe that most of the existing Pallas kernels are easily transferable to the new native TorchTPU serving stack but that does not mean that every existing kernel will transfer unchanged. Many things like wrappers, tensor layouts, and runtime integration etc may still need to be slightly adapted and validated.

Strong Ecosystem Collaboration with Google's TPU Team

vLLM and SGLang are the two major open-source production inference serving stacks. Inferact, RadixArk, and Red Hat are investing heavily in collaboration with Google to make the TorchTPU backend a first-class experience in both stacks. This community support is extremely important for externalizing TPU inference.

Source: Inferact

As of early September, TorchTPU remains in private beta and will be open-sourced around mid-October during the PyTorch Conference. We are excited to see it move TPUs toward a first-class experience in PyTorch, vLLM, and SGLang. Once TorchTPU vLLM and TorchTPU SGLang are open-sourced, we will move TPU benchmarking for InferenceX/AgentX from our fork to our public repo.

Source: RadixArk

TPU Inference Optimizations Deep Dive

the article, TPU architecture differs from GPU architecture and therefore requires its own set of optimizations to run models efficiently. Below, we describe the optimizations that improved (or in some cases, enabled) performance for Qwen3.5-397B and other models on Ironwood. Most of the Pallas kernel optimizations are transferable to the new native TorchTPU stack.

DP Attention Optimization

One issue that is not necessarily specific to TPU is how to effectively map parallelism from model architecture to hardware. For instance, Qwen3.5's GQA attention layer has 32 query heads but only 2 shared KV heads. With TP8 (eight logical devices), query computation can be evenly distributed across the 8 devices, with 4 query heads per device. The KV heads, however, do not divide evenly across those devices. Each KV head is shared by 16 query heads, meaning four devices need the same KV data to compute their local attention. When TP exceeds the KV-head count, standard practice is to replicate KV heads across TP ranks. vLLM already supports this. The TPU backend needed a compatibility fix to activate that existing behavior and avoid unnecessary All-to-All communication.

For higher-concurrency serving, the TPU backend also added support for combining eight-way attention data parallelism (DP8) with eight-way expert parallelism (EP8), simply referred to as "DP attention" or DEP8. Instead of splitting every request's attention across all eight devices, each device processes a different subset of requests and keeps both KV heads for those requests locally. The attention weights are replicated, but the KV caches hold different request histories rather than repeated copies of the same history. Meanwhile, the 512 routed experts remain distributed across the devices, avoiding replication of the much larger expert weight pool. Making this work required coordinating request assignment, recurrent-state slots, and block tables in the serving engine.

Source: SemiAnalysis

Optimizing Communications

With DP attention and EP, the TPU GroupedGEMM implementation all-gathers token activations and routing metadata before the experts run, then uses reduce-scatter to sum their weighted outputs and return each token's result to its attention rank. The backend originally gathered the selected expert IDs and routing weights separately. Google combined these into one all-gather, removing an extra collective whose latency can dominate the transfer time for such small arrays. The PR reports roughly 80 microseconds per layer saved in its DeepSeek-V3 measurements. While 80 µs may sound small, DeepSeek-V3/R1 has 58 layers that require this AllGather. Across those layers, the savings add up to around 4.64 ms per forward pass in isolation (80 µs × 58), reducing TPOT and increasing interactivity.

Source: SemiAnalysis

links to combine contributions within each chip before exchanging partial sums across chips. This two-device-per-chip layout and the chip-to-chip ICI network are covered in the next section. Double buffering lets the kernel transfer one chunk while accumulating another, overlapping local reductions and die-to-die transfers with the slower chip-to-chip traffic. Running the collective on SparseCore also frees TensorCore execution for other operations.

Source: SemiAnalysis

Source: Google and SemiAnalysis

In the diagram above, communication stages overlap, shortening end-to-end execution time. For instance, consider the point labeled t_1. At this time, the intra-chip DMA ScatterReduce (P1) has completed for MB (microbatch) 1. ScatterReduce across ICI dim 0 (P2.0) can now occur for MB0 *concurrently with* P1 for the subsequent MB. Against a baseline, this optimization results in 4.1 to 14.2% higher throughput on 8k1k across concurrency 64 to 512, including 8.5% at concurrency 256, and 26.1% on 1k8k at concurrency 512.

Thanks for reading SemiAnalysis! This post is
public so feel free to share it.

Share

The movement of this collective from TensorCore to SparseCore also means that additional work can be pipelined on the TensorCore while the collective runs. However, not every collective is always worth moving. In some cases, offloading a collective to SparseCores actually *worsens* performance. For example, for Qwen3.5 a threshold now decides when all-reduce and all-gather operations are offloaded to the SparseCore, **keeping small collectives on the TensorCore when they fit in VMEM and SparseCore offload overhead would make**

Optimizing MoE Routing and Expert Kernels

A mixture-of-experts layer produces irregular groups of tokens for each expert, and these ragged groups have to be reshaped into something the TPU's matrix units can consume efficiently. Some of the changes below are general improvements to the routing and grouped-matmul backend that benefit any MoE model. Others were measured directly on Qwen3.5.

SemiAnalysis is a reader-supported publication.
To receive new posts and support our work,
consider becoming a free or paid subscriber.

Type your email... Subscribe

The second version of the grouped matmul reworks how expert inputs are fed to the MXU. It removes redundant tile computation, sizes transfers to the number of valid rows rather than the padded maximum, triple-buffers expert weights so the next group's weights are already in flight while the current group is computed, and fuses group metadata generation into the kernel.

The irregular rearrangement of expert inputs was then moved onto the SparseCore. The SparseCore handles data movement, gathering each expert's tokens into contiguous groups, while the TensorCore is left to run the expert matrix multiplications. A subsequent rewrite improved memory-read pipelining and split the combine work across tokens and hidden dimensions. The PR reports a 12% increase in 8k1k serving throughput compared with the original SparseCore kernels, along with lower TTFT and TPOT.

Source: SemiAnalysis

A further optimization moved the top-k weight gather in the ragged gather-reduce path onto the SparseCore. Here, gather means reading selected entries from memory on one device, not an inter-device all-gather. Previously, the TensorCore gathered routing weights and source indices during preprocessing, limiting TensorCore/SparseCore overlap. Moving these gathers into the SparseCore kernel reduces TensorCore overhead from 29 µs to 14 µs and overall operation latency from 146 µs to 137 µs in a DeepSeek-V3 microbenchmark at batch 2k with 16-way expert parallelism.

For small batches, the general ragged path costs more than the work it is arranging. A dedicated small-batch permutation therefore builds one-hot matrices and uses ordinary matrix multiplication to permute tokens to their experts and unpermute the results. On the 8k1k workload, this improved throughput by 7.3% at concurrency 64 and 5.1% at concurrency 128.

A WIP change packs the expert ID and token index into a single sort key, so XLA performs a simpler sort while preserving the required ordering. Sort latency drops from 106.6 µs to 21.7 µs. The PR reports 8k1k serving gains of 0.6 to 8.5%, but the same change also enables an FP8 all-gather.

Optimizing Gated DeltaNet Pallas Kernels

Gated DeltaNet (GDN) is a recurrent computation. That is, at each step the running state is decayed, updated with a rank-one term, and projected to produce an output. Check out the following article for a deeper dive into linear attention mechanisms such as GDN:

The changes below show how its matrix operations, vector updates, and state transfers are scheduled across the MXU, VPU, VMEM, and HBM. Initial Qwen3.5 support added pure JAX implementations of causal Conv1D and GDN, connected them to TPU operator dispatch, and enabled recurrent-state caching. Later PRs optimized these implementations.

Source: GitHub

Another low-hanging fruit optimization is achieved by rearranging the algebra in the output projection calculation to overlap MXU and VPU work. Previously, the VPU first applied the rank-one update to the decayed state, and the MXU then multiplied the updated state by the query to produce the output. That dependency forced the MXU to wait.

Writing the decayed state as S, the correction vector as Δ, and the key and query as k and q, the output can be expanded:

Source: SemiAnalysis

calculated separately using k·q, a scalar dot product per head, followed by a small scaled-vector addition. This removes the state update from the MXU's dependency path. Reported 8k1k throughput gains were 2.79% at concurrency 64 and 4.48% at concurrency 512.

The next change reduces vector-register spills by slicing Q and K inside the decode loop so fewer values stay live at once. The decode-64 kernel becomes roughly 20% faster, but end-to-end gains are smaller, at 0.8% on 8k1k and 3.8% on 1k8k at concurrency 512, since the kernel is only one part of a decode step.

Asynchronous state transfers overlap DMA with computation using double buffering. The extra VMEM consumed by the second set of buffers initially regressed data-parallel attention. Reusing existing scratch buffers and shortening the lifetimes of temporaries recovered that capacity and removed the regression. The reported 8k1k throughput gain is 11.3% at concurrency 512.

GDN v3 fuses Conv1D and GDN into a single kernel to cut HBM round trips, improves prefill layouts, and unifies mixed prefill/decode execution into one path. Reported kernel-level speedups are 1.41× for decode, 1.60× for prefill, and 2.14× for mixed batches. These measurements cover the kernel alone and do not establish end-to-end serving gains.

Source: SemiAnalysis

Significant performance gains can be achieved simply by improving the overlap between operations within a kernel!

Managing Hybrid State and Paged Attention

State allocation, storage precision, attention page size, and physical data layout all determine how much HBM is actually usable and how efficiently attention runs.

SemiAnalysis is a reader-supported publication.
To receive new posts and support our work,
consider becoming a subscriber.

Type your email... Subscribe

Batched ragged paged attention batches sequences together, precomputes page metadata, and triple-buffers to improve pipelining and reduce padding. This change provides shared attention-backend groundwork. The PR's workload examples use Qwen3-32B, so its measurements should be kept separate from the Qwen3.5-397B results.

Recurrent state is now allocated compactly, with roughly one slot per active request instead of `num_blocks` slots for every layer group. In the reported configuration, this reclaims about 76 GiB of HBM and expands the attention block pool by 71%, improving 1k8k output throughput by 18% at concurrency 64.

Storing the recurrent state in BF16 halves its HBM footprint while keeping the arithmetic in FP32 inside VMEM. This saves memory capacity and transfer bandwidth while preserving FP32 arithmetic. The reported 1k8k throughput gain is 15% at concurrency 512.

Removing the old hybrid page-size alignment constraint lets batched attention use a suitable power-of-two page size, including 256 tokens. 1k8k throughput improves by about 7% at concurrency 512. This applies to the non-prefix-caching path and is distinct from the aligned checkpoint mode introduced later.

Source: SemiAnalysis

Unnecessary KV-layout copies are avoided by selecting the reshape path based on KV-head count, rather than imposing a layout constraint that only other shapes need. Qwen3.5 8k1k throughput improves by about 4.1% at concurrency 512.

Reducing Padding and Low-Concurrency Overhead

When only four or eight requests are in flight, a configuration tuned for hundreds of concurrent requests wastes work. Compiled shape buckets are too large, metadata is sized for the configured maximum, and padding tokens trigger dummy expert work. The InferenceX operating points sit at exactly these low concurrencies, which motivated the following changes.

Type your email... Subscribe

Request metadata is now bucketed by the number of active requests instead of always using the configured maximum. In the 8k1k concurrency-64 test, GDN scheduling overhead falls from 283 µs to 97 µs and throughput rises from 2,328 to 2,516 tokens/chip/s.

Source: SemiAnalysis

Explicit Qwen3.5 tuning for InferenceX shrinks rotary tables, sizes sequence limits per DP rank, and adds a dedicated attention bucket for concurrency four. The combined gains at concurrency four are 13.3% on 8k1k and 15.5% on 1k1k.

A further round of low-concurrency tuning switches to TP8 attention plus expert parallelism, reduces the minimum token bucket, and routes padding tokens to expert zero so they do not trigger additional expert weight loads. Combined 1k1k gains are 22.9% at concurrency four and 18.1% at eight. On 8k1k, throughput rises by 9.2% at concurrency four but falls by 5.3% at concurrency eight.

Enabling Prefix Caching for Hybrid Models

The gains above were measured on random-input benchmarks, where nothing is shared between requests. Prefix caching is crucial for agentic and multi-turn workloads, which reuse long system prompts and conversation histories. For a hybrid model, a cached prefix must retain both its KV blocks and the GDN recurrent state at the end of the prefix. That recurrent state is normally overwritten as soon as the request continues, making prefix caching harder than for "regular" attention models.

Hybrid prefix caching with DP support resolves this by giving GDN separate slots for reading a checkpoint and writing the live state. Continuing a request therefore no longer overwrites the checkpoint associated with its cached prefix. State addresses are derived from the block table, the same structure that already locates KV blocks. Checkpoints are taken at an aligned cache granularity so a saved state always lines up with a KV block boundary. This mode needs a full checkpoint pool rather than the compact per-request

Source: SemiAnalysis

Lane Layout and Pipelining Depth in Paged Attention

Two hardware facts shape how a KV cache is laid out on TPU. We discuss TPU hardware in more detail in the following section. First, the vector unit works on tiles whose last dimension is 128 lanes wide, so any array whose trailing dimension is smaller than that is padded up. Second, a Pallas kernel hides HBM latency through double-buffering. It fetches the next block while computing the current one. Both blocks must fit in VMEM, so the size of the compute block determines how deep the prefetch can be. Both constraints were costing capacity and throughput.

In the batched attention kernel, the KV cache packed keys and values along the head dimension, and for FP8 that packing factor is four. A model with a single KV head per device only has two things to pack, so half of every tile was wasted on padding. A sequence-on-lane layout fixes this by putting the page's tokens on the 128-lane axis and the head dimension on the sublane axis instead. Usable KV pages double (from 5,141 to 10,283 in the reported configuration) and the head dimension only needs to be a multiple of 32 rather than 128, which allows models with a head dim of 64 to use the kernel as well. The layout costs about 3% in per-token latency at low concurrency, but at concurrency 128 on 8k1k the extra capacity lifts throughput 16.5% and cuts median TTFT by 95%, because requests no longer wait for KV space.

Source: SemiAnalysis

The RPA v3 block-size heuristic had a similar blind spot. During v7x decode, it set the KV compute block equal to the KV fetch block, around 16k tokens, which left almost no VMEM for the prefetch buffer. Splitting the two block sizes kept the KV fetch block at 16k tokens while reducing the compute block to 4k. This gave the pipeline room to run ahead of the MXU and increased decode throughput from 64.9k to 96.3k tokens per second, a 49% gain reproduced across four runs on Qwen3-0.6B. Throughput followed a clean inverted-U curve across the block-size sweep. The tuned-parameter table could not represent the improvement because it stored a single block size per shape, which is why the change first landed as an environment override. A follow-up extends the table to hold the fetch and compute sizes separately.

Source: SemiAnalysis

Networking Explained

As we said in the TPU King article, Google's cost-per-token advantage on inference is a result of co-design. Instead of focusing on maximum single-chip performance, Google designs the compute die, chip-to-chip fabric, and compiler together, allowing it to optimize computation and communication as a single system. That joint optimization helps explain Ironwood's performance per dollar. The following sections examine how the chip and network work together. Anthropic is a big fan of TPUs and uses them heavily for training.

The Ironwood Chip

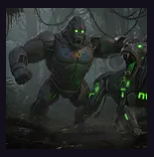
TPUv7 Ironwood breaks from the "MegaCore" convention that defined TPU v4 and TPU v5p, where two physical cores were fused into a single logical accelerator sharing one memory space. Ironwood instead has two separate compute dies, each running its own independent logical device. These dies are joined by a high-bandwidth die-to-die link rather than a unified memory fabric. JAX and other frameworks now expose them as two distinct devices per chip. Each Ironwood chip carries 2 TensorCores and 4 third-generation SparseCores. The SparseCores accelerate embedding lookups and other sparse operations that would otherwise choke a dense matrix engine.

Source: Google

generation with native FP8 hardware support, whereas prior generations had to emulate FP8 in software.

Source: SemiAnalysis

Read more on Ironwood in our TPUv7 article



TPUv7: Google Takes a Swing at the King

DYLAN PATEL, MYRON XIE, AND 10 OTHERS · 2025年11月28日

Read full story →

The MXU and Why Shapes Matter

The Matrix Multiply Unit, the engine that actually does the multiplying, is a systolic array: a two-dimensional grid of multiply-accumulate cells. Weights are loaded into the array and held stationary. Activations stream in from the edge, and partial sums ripple through the grid cell by cell, accumulating as they go. The finished result streams out the other side without ever touching memory mid-computation. Every TPU generation through v5 used a 128x128 MXU, good for 16,384 MACs per cycle. Starting with TPU v6e and carrying through to Ironwood, the array doubled to 256x256, or 65,536 MACs per cycle, delivering 4x more FLOPs per cycle than the previous design.

Please read the scaling book (aka the ML systems bible) for excellent top-tier information on TPUs and scaling in general.

SemiAnalysis is a reader-supported publication.
To receive new posts and support our work,

Type your email... Subscribe

The catch is that a bigger systolic array is only free if you can keep it full. Matrix dimensions need to be padded up to at least the MXU's side length in both directions, 128 on older generations or 256 on v6e and v7, and the XLA compiler dutifully pads any smaller axis to fill the tile. Every padded cell still occupies a MAC unit for that cycle, multiplying by a zero that contributes nothing to the answer. Llama 3 8B illustrates this with an attention head dimension of 128. On Ironwood's 256x256 MXU, that head dimension is exactly half the array's native width, which caps the two attention matmuls at a maximum of 50% MXU utilization (not 75%, because only the head dimension is 128).

The implication runs deeper than one model's attention layer. Architectural choices that used to be arbitrary hyperparameters (such as head dimensions, the KV head count that survives after tensor-parallel sharding, and MoE expert widths) increasingly need to be made with TPU tile geometry in mind, because a shape mismatch there is a direct tax on delivered throughput regardless of how good the rest of the stack is. Kernel authors also need to account for this explicitly. Production attention kernels like ragged paged attention use explicit packing dimensions to reduce padding when XLA's default tiling produces an inefficient layout.

TPUs Are Picky

GPU matrix cores consume small tiles, so a wide range of head dimensions, expert widths, and post-sharding KV head counts all land near peak. A dimension of 64 instead of 128 pays close to nothing on an H100 or a B200 and gets a cheaper attention layer for it. GPU architecture allowed researchers to prioritize evaluations performance without a loss in inference performance. But for TPUs, this hyperparameter becomes a tradeoff

As previously mentioned, these choices are not free on a 256-wide systolic array. By the same arithmetic that caps Llama 3 8B's attention matmuls at 50%, a head dimension of 64 caps them at 25% before a single line of kernel code is written. gpt-oss ships at 64. DeepSeek's MLA splits its query/key dimension into 128 plus 64 for a total of 192, ungainly against any power-of-two array and worse against a wide one.

The consequence is that bring-up cost varies enormously, and it correlates poorly with how popular a model is. A model whose shapes fall out cleanly needs scheduling, serving, and tuning work, which is measured in weeks. A model that fights the tile geometry needs new kernels before it is even at parity, let alone winning on performance per dollar. Kernel engineers are finite, so sequencing externalization toward models where the hardware is not starting a lap down is the rational move, and we would expect the same behavior from any accelerator vendor in this position.

Torus Topology

Inference and training at scale run across thousands of chips that need to talk to each other constantly. TPUs connect P2P over a custom network called ICI (Inter-Chip Interconnect). ICI bypasses the host CPU entirely, allowing chips to exchange activations and gradients directly instead of routing them through PCIe and a general-purpose NIC. The topology behind ICI has evolved generation over generation: TPU v2 and v3 used a 2D torus with each chip wired to 4 neighbors, and starting with TPU v4 and v5p, Google moved to a 3D torus with each chip wired to 6 neighbors along the +/-X, +/-Y, and +/-Z axes. Ironwood (TPUv7) keeps that 3D torus. The fundamental building block is a 4x4x4 cube of 64 chips, sized to map cleanly onto one physical server rack.

Source: SemiAnalysis

trick that makes Pac-Man's maze feel smaller than it is. Google pushes this further with a "twisted torus," a Mobius-strip-like wraparound that shaves the average hop count down even more. To scale beyond a single 4x4x4 cube, Google stitches cubes together using Optical Circuit Switches, which preserve the wraparound property across a much larger reconfigurable topology, scaling all the way up to Ironwood's full 9,216-chip superpod and its 42.5 FP8 exaflops of aggregate compute. The operational payoff of OCS is that Google can physically rewire around a failed link or a dead chip in seconds, using mirrors rather than sending a technician to re-splice copper in a live data center.

This scale-out design was revolutionary at the time. Before NVL72 racks, models that could not fit within a single 8-GPU node had to use pipeline parallelism because inter-node InfiniBand was slow. On a TPU pod, the ICI torus gives NVLink-class bandwidth across the entire pod, up to 8,960 chips on v5p. With this bandwidth, a DSV3-scale model can be sharded across the whole pod using tensor parallelism, expert parallelism, and data parallelism (FSDP-style weight sharding), without splitting layers across pipeline stages.

Although a TPU torus has more hops than a single-hop NVLink switched design, our upcoming CollectiveX/NetworkingX results show that, in many cases, the TPU torus has lower latency than a single-hop NVSwitch for small EP messages.

Looking Ahead: TPUv8i's Boardfly Network

Google's newly announced eighth-generation TPU lineup has two purpose-built chips: TPU 8t for training and TPU 8i for inference. This is the first time Google has separated training and inference into distinct architecture designs rather than shipping one architecture tuned for both. TPU 8t keeps the 3D torus lineage alive for scale-up, but TPU 8i replaces the torus with a new topology called "Boardfly." The name nods to dragonfly-style high-radix network designs long used in supercomputing. Boardfly uses a flatter, hierarchical fabric of high-radix switches rather than a nearest-neighbor mesh. The Boardfly topology affects networking attachment capex per chip.

The payoff is a network diameter cut by more than 50% versus a similarly sized 3D torus, roughly 16 hops down to about 7 hops at comparable scale in the 1,024 to 1,152-chip range. Fewer hops mean materially lower tail latency on collective operations, and that matters enormously once you are routing tokens across MoE layers or running multi-turn agentic workloads where every extra hop compounds into user-visible latency. TPU 8i backs this up with 19.2 Tb/s of ICI bandwidth, double the prior generation, and 384 MB of on-chip SRAM, 3x the previous generation, sized specifically to hold the KV cache of reasoning and agentic models on-chip rather than round-tripping to HBM.

Source: Google

None of this hardware headroom pays off on its own, though. Turning a lower-diameter network and a bigger on-chip cache into cost-per-token gains still requires the software stack to catch up. The next section covers the roadmap for doing so, including speculative decoding, prefill-decode disaggregation, and the broader model support needed to take full advantage of TPUv8 in production.

Next Steps in Laying the TPU Foundation

These preview results establish a starting point for further optimization on the same hardware. We expect performance to improve as TPU externalization continues, with substantial work still needed across the software stack.

Enabling & Optimizing Spec Decoding (MTP)

The first area the Google team needs to focus on is on optimizing speculative decoding. A cheap, small drafter guesses several tokens ahead, the main model verifies all of them in a single forward pass, and any guess that doesn't match is thrown away. Everything that survives is identical to what the model would have produced on its own. Speculative decoding is lossless, with zero quality degradation.

weight read is the dominant cost, and the cost barely changes whether you verify one token or five. The matrix units sit mostly idle while the memory system does all the work. Speculative decoding spends that idle compute to amortize one very expensive weight read across several tokens, which is why methods that draft multiple tokens at once, like MTP or DSpark, are so cheap and so effective.

Source: vLLM

Prefill-Decode Disaggregation & KV-cache Offloading

Prefill-decode (PD) disaggregation is another optimization the TPU team is working to externalize. It separates prefill and decode across distinct TPU pools, allowing each pool to be tuned and scaled independently to match the workload.

Source: DistServe

in mid-d and the open-sourcing of TPU-Sync (formerly TPU-Taiden), Google's disaggregated KV-cache transfer library. TPU-Sync works natively with JAX and the native TorchTPU stack, and can perform zero-copy transfers by extracting native PJRTBuffer hardware descriptors. With disaggregated PD, we believe TPUv7 can be a strong competitor to GB200/GB300 and even beat them on performance per dollar.

SemiAnalysis is a reader-supported publication.
To receive new posts and support our work,
consider becoming a subscriber.

Type your email... Subscribe

TPU-Sync also supports native TPU KV-cache DRAM offloading, which is needed for large models and medium-to-large batch sizes, where HBM can no longer hold the KV cache for all users. It is great to see Google externalizing its DRAM offloading optimizations to the public as well.

Source: GitHub

pooling support. We believe Mooncake Store support will be implemented using the primitives in tpu-sync.

Source: GitHub

With P2P pooling, the KV-cache storage on each TPU host is aggregated into a single logical memory pool, so a TPU on any server can access KV cache from any other server. This unifies memory contributions from multiple nodes into one shared logical pool.

Source: Mooncake

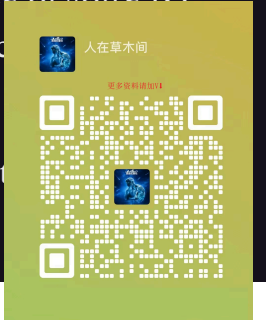
Furthermore, Mooncake Store pools NVMe storage from multiple servers into a single logical pool in addition to supporting traditional distributed filesystem backends like WEKA/Vast.

AgentX TPU

KV-cache offloading is especially important for long-context, multi-turn agentic workloads, where optimized KV-cache storage can enable high KV-cache hit rates.

At a high level, an agentic workload is characterized by four elements:

1. Multi-turn: a session includes tens or hundreds of interactions between the user and assistant, compared with a handful in a chatbot scenario. These workloads combine long contexts and high prefill reuse with sub-agent bursts and numerous tool calls.
2. Long context: system prompts, tool definitions, and the large number of turns make context accumulate quickly.
3. High prefix reuse: since the conversation progresses linearly, where output from turn n-1 is concatenated to turn n (typically), most context can be served from KV cache rather than recomputed (this depends on the amount of storage available to store KV tensors). As n grows, the ratio of cached input relative to uncached typic towards 1.
4. Sub-agent bursts: a session launches multiple short-lived sub-agents wit context, which create bursty KV-cache patterns.



AgentX too!

Source: DeepSeek, SemiAnalysis

Ironwood has no native FP4, so for now the fair apples-to-apples comparison is against FP8 Blackwell: FP8 vs FP8 carries no quality loss, whereas FP4 vs FP8 introduces a quality difference on the FP4 side. Google's TPUv8i does have native FP4 acceleration, so when we bring up TPUv8i on InferenceX/AgentX, we will compare FP4 to FP4.

SemiAnalysis is a reader-supported publication.
To receive new posts and support our work,
consider becoming a subscriber.

Type your email... Subscribe

We strongly believe that building native vLLM and SGLang support on native TorchTPU's stable foundation is the right direction for TPU externalization. TorchTPU is meant to replace the previous TorchAX stack, which will soon be deprecated. Google will next work to enable Kimi K3 and GLM5.3 on single-turn and/or agentic workloads like AgentX, along with Google's own open-weight models such as Gemma4. Once a few models are running on the TorchTPU stack, adding new ones will become far easier, and TPU support will land much closer to day 0.

TPU Cost Effective Total Cost of Ownership

Our full BOM and TCO estimates for TPUv7 are enumerated in the SemiAnalysis AI TCO Model. A structural overview is below.

This post is for paid subscribers

Subscribe

Already a paid subscriber? Sign in

← Previous

© 2026 Dylan Patel · Privacy · Terms · Collection notice

Start your Substack | Get the app

Substack is the home for great culture