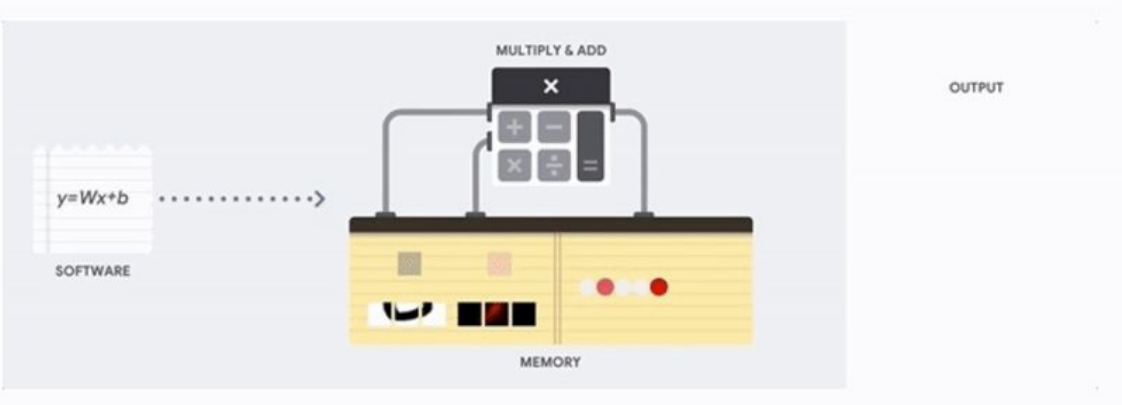
Building AI Supercomputers

1. TPU 101: Google's TPU Architecture
2. The Software Layer: JAX & XLA
3. Orchestration
4. Industry references & “From the field”

Erik Saarenvirta (esaarenvirta@)
Staff Technical Solution Consultant
Google

TPU 101

Refresher: CPUs



- **Latency-Optimized**: Designed to execute single threads as fast as possible.
- **Complex Control Logic**: Focuses on branch prediction and out-of-order execution.
- **Large Caches**: Heavy reliance on L1/L2/L3 caches to hide memory latency.
- **The ML Bottleneck**: Too much silicon spent on control logic instead of raw math.

Refresher: GPUs

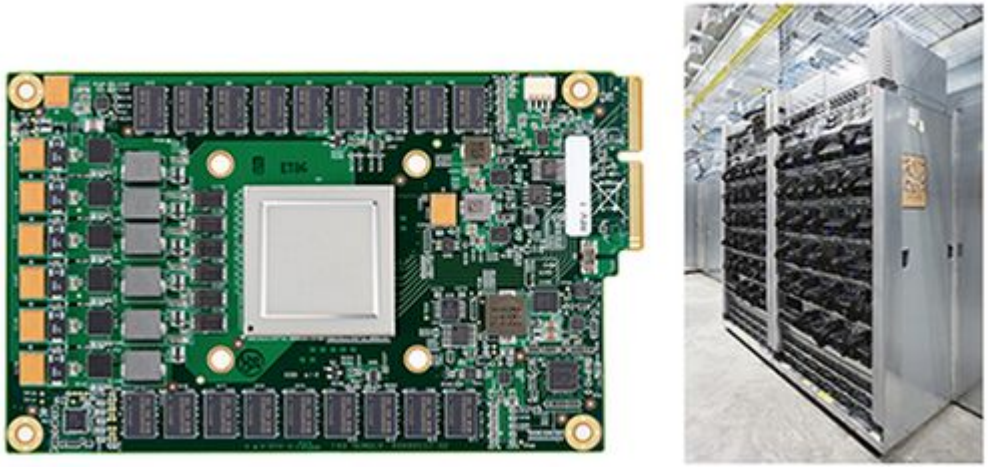
- **Throughput-Optimized**: Processes thousands of simultaneous operations.
- **SIMT Architecture**: Single Instruction, Multiple Threads.
- **High-Bandwidth Memory (HBM)**: Feeds massive data to thousands of ALUs.
- **The ML Shift**: Graphics rendering hardware perfectly mapped to neural network matrix math.

Note: This animation is designed for conceptual presentation purpose only, and does not reflect the actual behavior of real processors.

Google Cloud

Domain Specific Architectures

Tensor Processing Unit

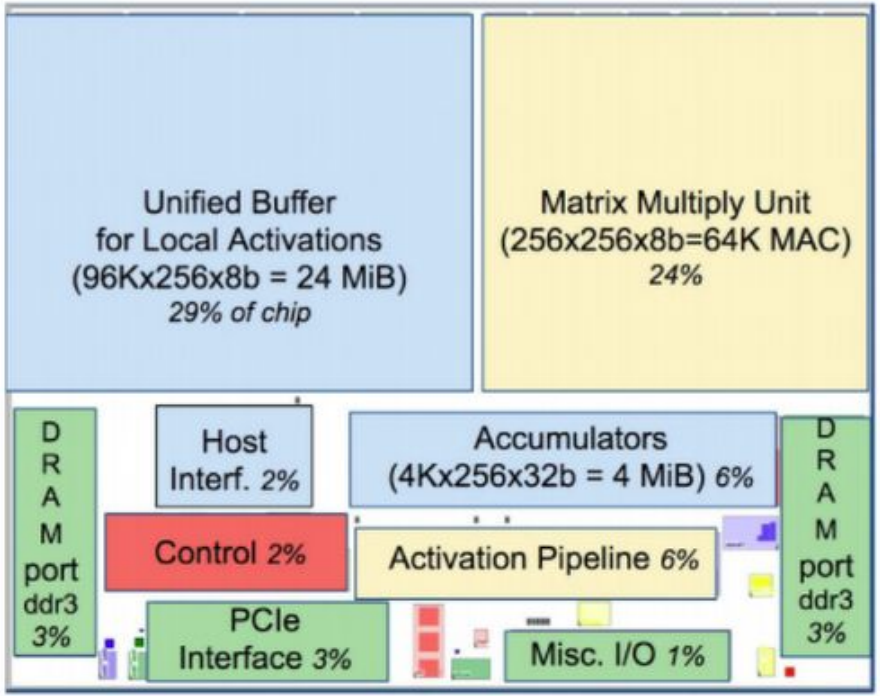


Google's first Tensor Processing Unit (TPU) on a printed circuit board (left); TPUs deployed in a Google datacenter (right)

- **GPUs**: Powerful, evolving, and dominant in AI training and inference.
- **The Custom Chip (ASIC)**: Massive value in building bespoke silicon to control the full hardware-software stack and optimize for specific workloads (e.g Gemini)
- **Purpose-Built**: TPUs are designed *strictly* for neural networks.
- **Density over Flexibility**: Stripping away general-purpose hardware (ray tracing, caches) for pure matrix math.

TPU V1

Tensor Processing Unit



Google Cloud

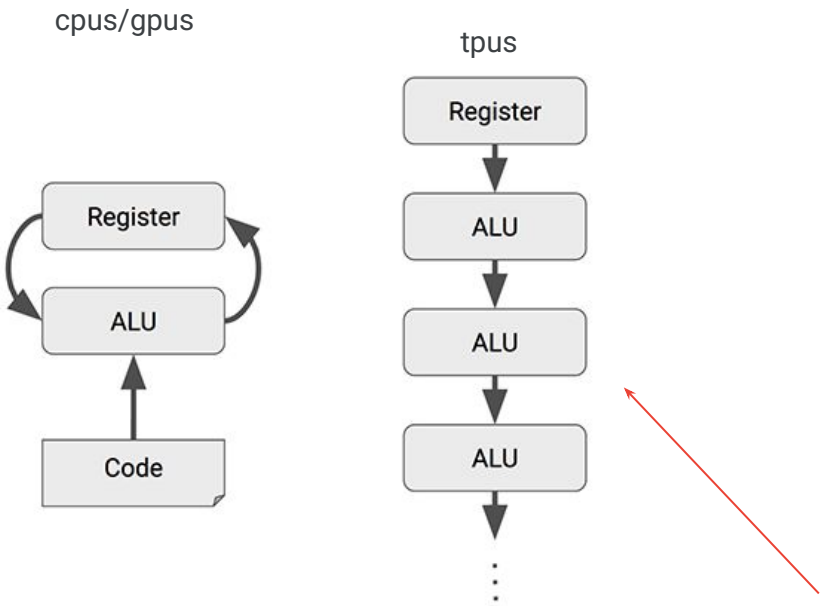
Reference: Google Cloud Blog - First TPU

- CISC Instruction Set: High-level AI instructions (e.g., MatrixMultiply Activate).
- 8-bit Quantization: Focuses on integers over floats to save power and space.
- Software-Managed Memory: 24MB SRAM Unified Buffer (no hardware caches).
- Generational Leap: 15–30X higher performance than contemporary alternatives (CPU & GPU) at launch.

Note: Notice the lack of space needed for control logic

Systolic Arrays

MXU Architecture



- **TPU Design (Spatial Compute)**: Data flows through a 2D ALU grid (Systolic Array), maximizing *data reuse* since no returning to a register
- **GPU Design (Latency Hiding)**: Relies on extreme multi-threading and High-Bandwidth Memory (HBM).
- **Where the Silicon Goes**:
  - **GPUs**: Massive register files & complex thread schedulers.
  - **TPUs**: Contiguous grid of ALUs for raw math throughput.
- **Hardware Convergence**: NVIDIA's *Tensor Cores* act as specialized mini-systolic arrays.

Wires connect only spatially adjacent ALUs + they can only perform multiplication and addition

Google Cloud

Systolic Arrays

MXU Architecture

Multiplying an input vector by a weight matrix with a systolic array

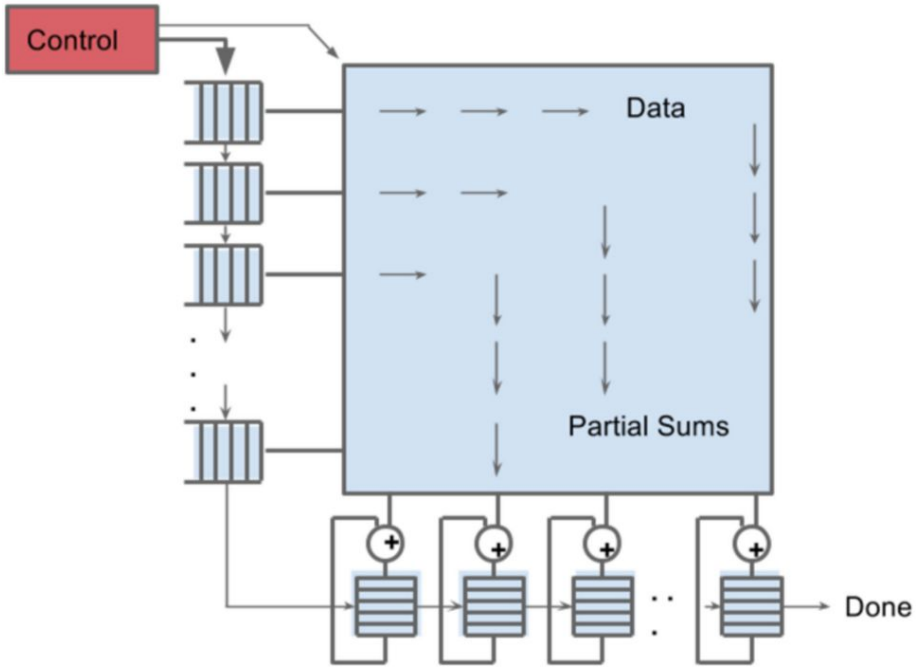
Matrix multiplication is just a series of multiplying and adding numbers

Efficiency⟶ ALUs perform only multiplications and additions in fixed patterns ⟶ only wired to spatially adjacent so short lived

Systolic ⟶ data flows in waves, like a heart pumping blood

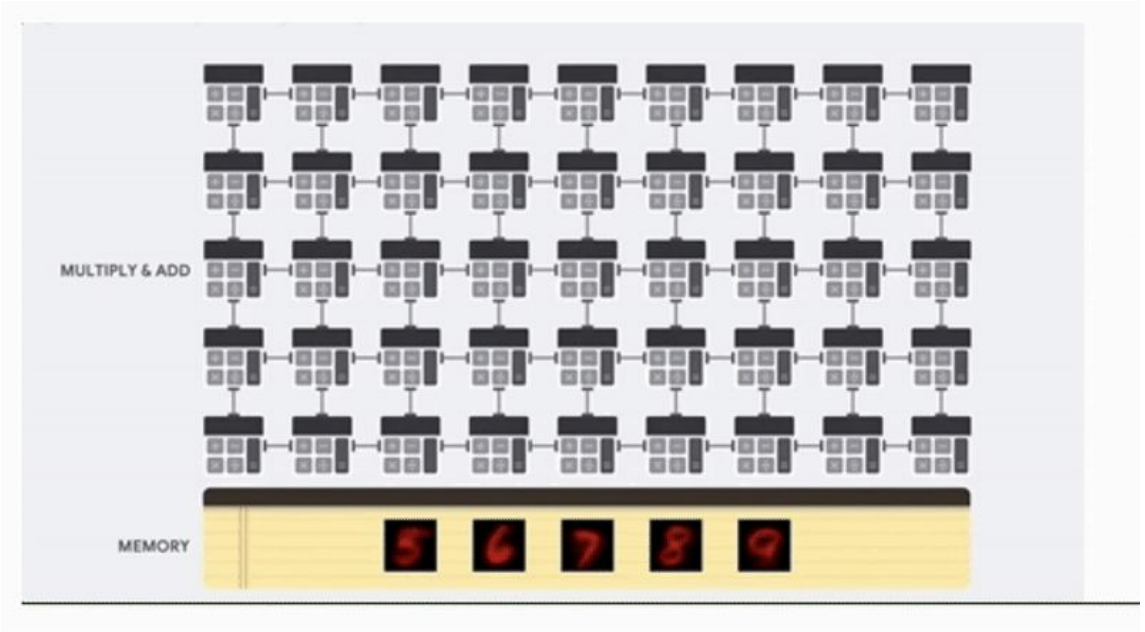
Systolic Arrays

MXU Architecture



- TPU v1 MXU was 256 × 256 = total 65,536 ALUs = 65,536 multiply-and-adds for 8-bit integers every cycle.
- A TPU runs at 700MHz ⟶ Can compute 65,536 × 700,000,000 = 46 × 1012 multiply-and-add operations or 92 Teraops per second (92 × 1012) in the matrix unit.
- A single MatrixMultiply cycle ⟶ 100s of 1000s of ops ⟶ Intermediate results passed between all 65K ALUs without memory access

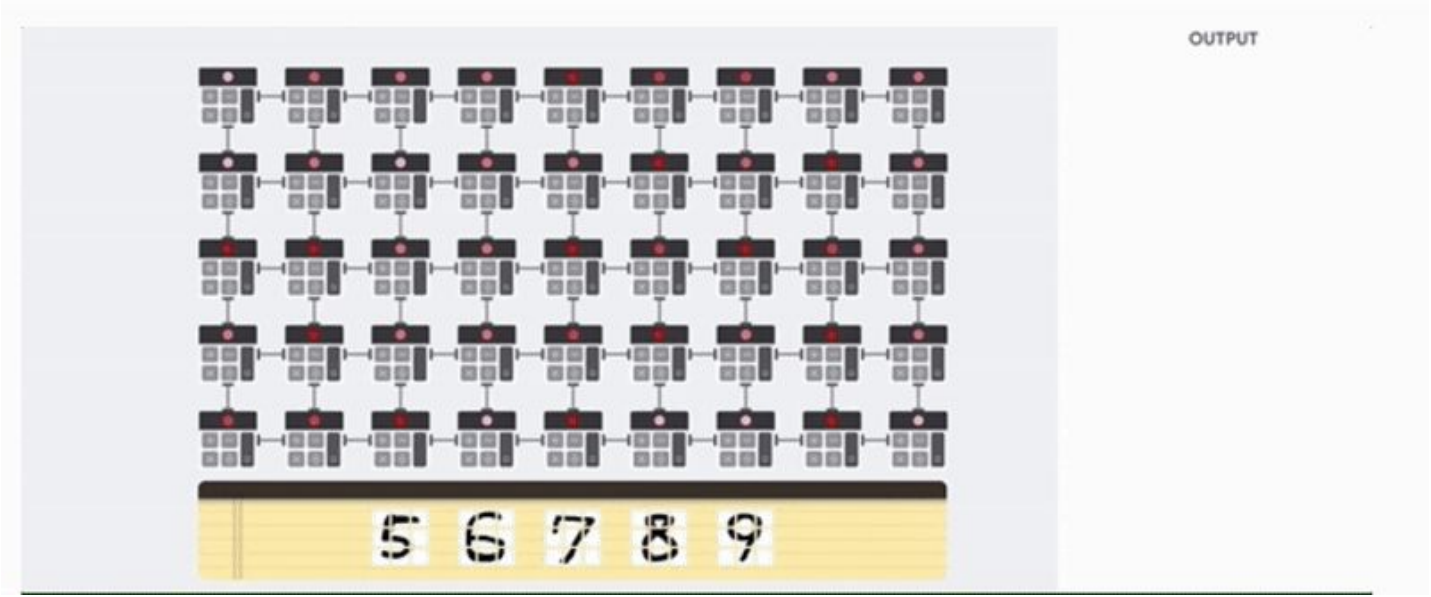
Returning to our visualizations: TPU edition



Loading model parameters into an MXU

Google Cloud

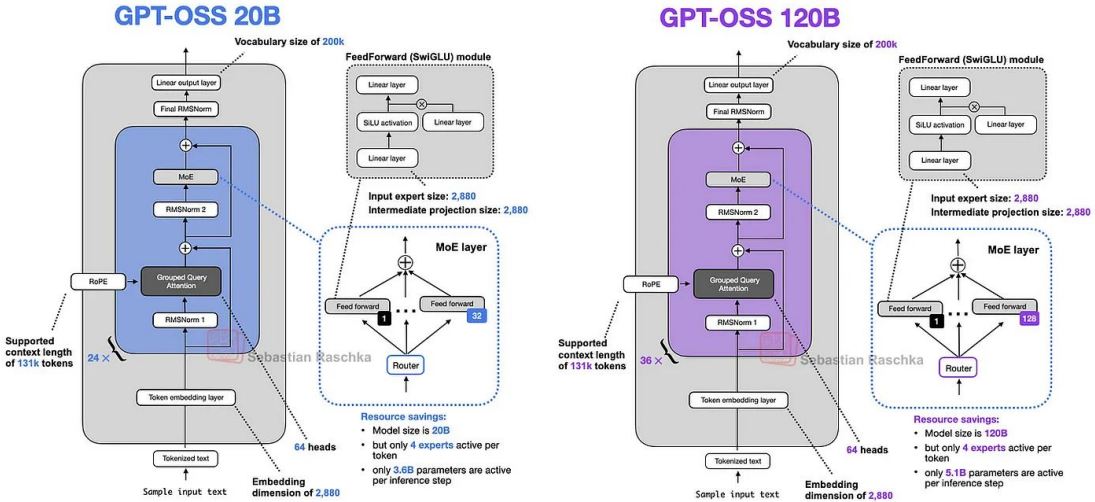
Returning to our visualizations: TPU edition



Load data, perform a matmul (params x data) in the MXU

Hardware Aware Design

Tensor Processing Unit



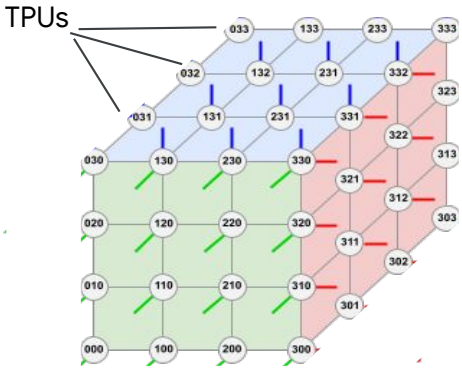
Note a head dim of 64 → head dim sharding very common with LLMs → 64 < 256

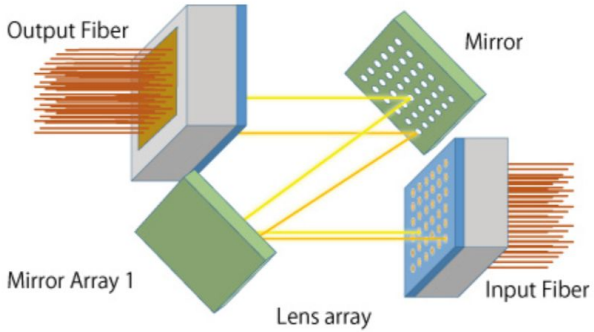
- **Model dimensions** (like 128, 256) aren't just mathematically convenient; they are silicon-driven.
- **The MXU Constraint**: Systolic arrays are fixed, rigid physical grids (e.g., 256x256 or 128x128).
- **The Padding Tax**: If your matrix dimension doesn't perfectly tile the MXU, the compiler pads the rest with zeros.
- **Wasted Compute**: Padding means the chip is burning power multiplying by zero, destroying your Model FLOPs Utilization (MFU).
- **Design your models** to be hardware aware (Gemini, Gemma) to maximize MFU

Google Cloud

The Network is the Computer

Tensor Processing Unit ICI (Inter-Core Interconnect)





- **The Multi-Chip Bottleneck**: A single TPU is fast, but LLMs require thousands. Traditional ethernet networks introduce fatal latency.
- **What is ICI?**: Google's bespoke, high-bandwidth, direct optical network connecting TPUs without going through traditional networking switches.
- **The Topology**: TPUs are wired in a 3D Torus topology (a massive wrapping cube of chips).
- **Why it Matters**: ICI allows our software frameworks (coming up) to treat 4,096 physically distinct chips as one massive, logically unified supercomputer with predictable latency. Can easily route around failed chips

Google Cloud

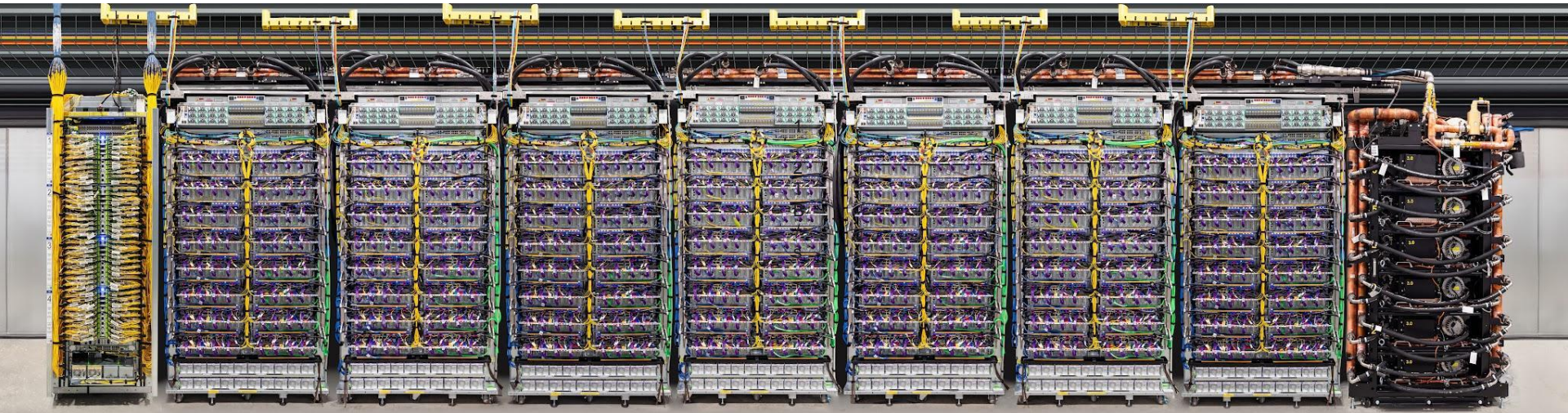
Optical Circuit Switch Paper

9,216

chips

42.5

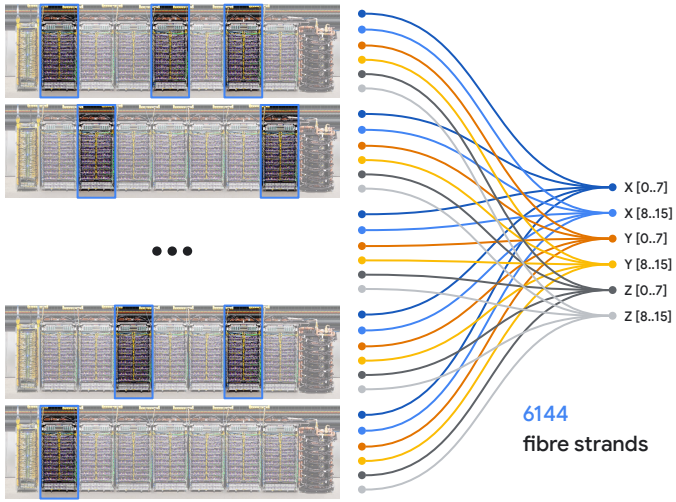
Exaflops per pod



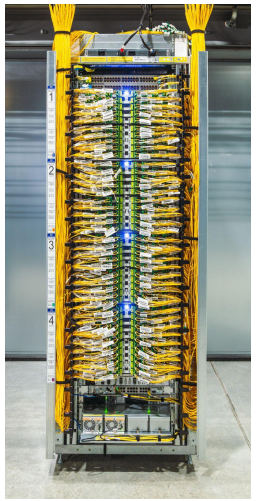
Optical circuit switch in action

Ironwood TPU pod

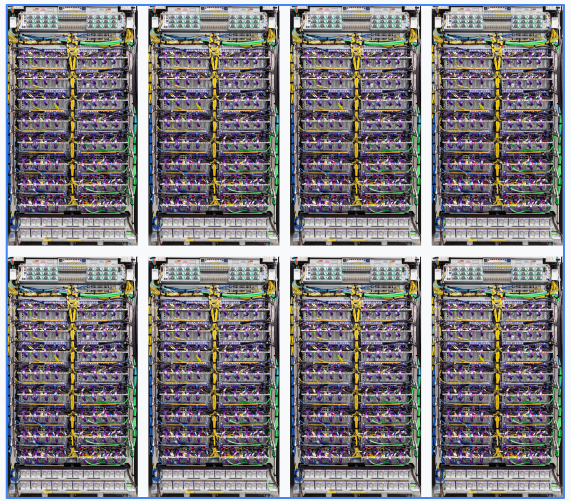
Suppose you want a 16 x 8 x 4 configuration
This requires 8 racks put together in a particular way.



Racks are physical 4 x 4 x 4 cubes



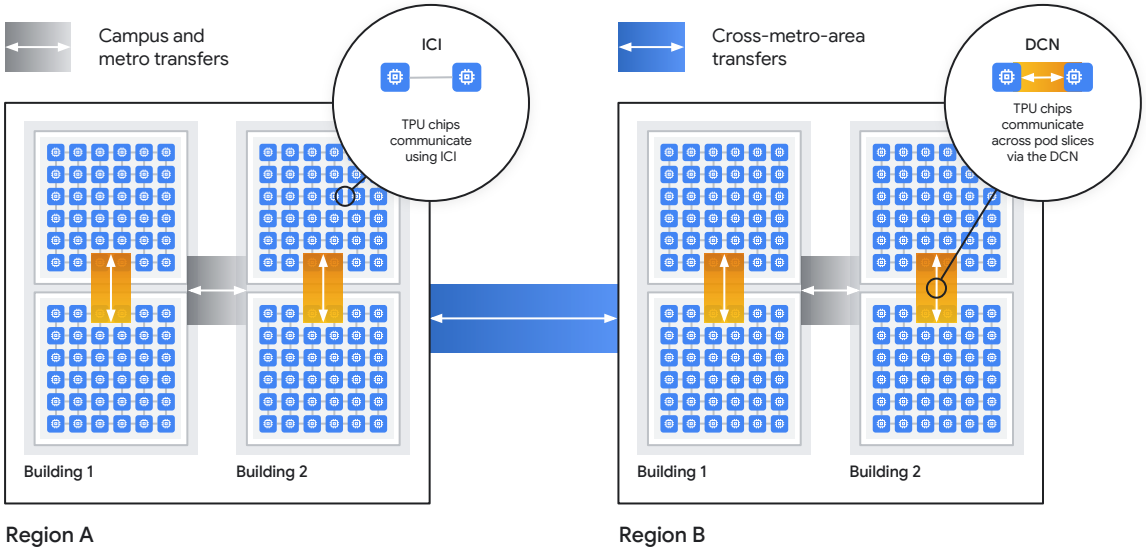
x48 OCS



Logical 16 x 8 x 4 chip configuration

Data center networking fundamentals

Fifth-generation Jupiter data center network architecture



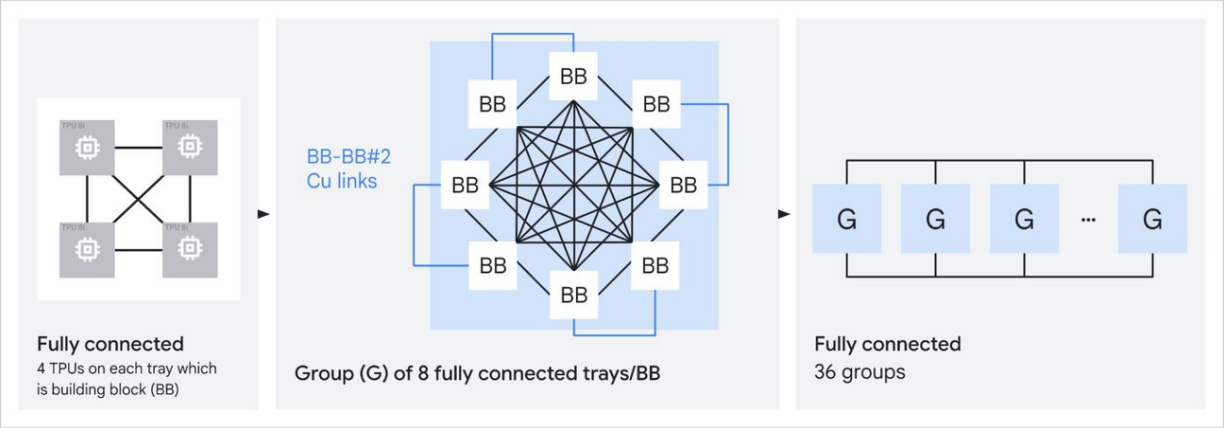
- Anything placed anywhere among 100k+ TPUs
- Predictable, low latency
- Software-defined and systems-centric
- Incremental evolution and dynamic topology
- Traffic engineering and application-centric QoS
- Client with JAX+Pathways

Into the future

TPU v8t and v8i

[Read the technical deep dive here]

8t is similar to Ironwood (v7) with improved specs, training focused. 8i moves in a new direction for inference and RL (sampling) with a “Boardfly” high-radix topology instead of a 3D torus. Deployed in pods of 1152.



Training = Throughput: The 3D Torus remains the standard (scaling to 9,000+ chips per pod) for efficiency. All-to-all hidden behind huge batch sizes and sheer FLOPs required

MoE inference bottleneck: Because experts are sharded across the pod, next-token prediction forces latency-sensitive, all-to-all network traffic

Network hops: In a 1,024-chip 3D Torus, traversing the rings requires up to **16 hops** (8/2X + 8/2Y + 16/2Z = 16).

The Boardfly Solution: Uses a high-radix topology to flatten the network diameter down to just **7 hops**.

The Impact: This **56% reduction** in tail latency ensures TPU 8i compute isn't left sitting idle waiting for cross-pod data routing.

Google Cloud

Bridging the gap between software and hardware

JAX as a framework

```
import jax, jax.numpy as jnp

# 1. NumPy-like syntax
x = jnp.array([2.0, 3.0, 4.0])

# 2. Autograd (Automatic Differentiation)
def f(x): return jnp.sum(x**3)
df_dx = jax.grad(f) # Returns a derivative function

# 3. XLA Compiler (Just-In-Time)
fast_f = jax.jit(f) # Compiles for the accelerator
```

- **What is JAX?**: A curated set of interoperable libraries (NumPy + Autograd + XLA) for high-performance ML research.
- **Core Philosophy**: Achieve *performance, flexibility, and scalability* entirely through pure function transformations.
- **The Engine (XLA)**: Compiles highly optimized code for the target hardware. It understands network topologies (like the 3D Torus), abstracting a 4,000-chip cluster into a single logical computer for the user.
- **True Portability**: Write once, run anywhere. Execute the exact same code across CPUs, GPUs, and TPUs often without modification.

Flexible Parallelism - Library-based vs Compiler Driven

JAX as a framework

- The PyTorch Approach (Library-Based): Scaling requires wrapping your model in specific library objects (e.g., `model = DDP(model)` or `FSDP`). The framework manages communication under the hood at runtime.
- The JAX Approach (Compiler driven): JAX uses a unified SPMD (Single Program, Multiple Data) model driven entirely by the compiler.
- Why JAX is More Flexible:
  - In PyTorch, switching from Data Parallelism to Tensor Parallelism requires changing frameworks or rewriting significant code.
  - In JAX, you simply change the *sharding annotations* on your arrays. The XLA compiler mathematically proves the new layout and rewrites the distributed program from scratch.

```
# 1. Define device topology (e.g., an 8-device cluster)
sharding = PositionalSharding(devices).reshape(2, 4)

# 2. Tell JAX how to distribute the tensor
sharded_x = jax.device_put(x, sharding)

# 3. Perform math. XLA handles all networking!
z = jnp.dot(sharded_x, sharded_y)
```

Hello world parallelism example

```python
import jax
import jax.numpy as jnp
from jax.sharding import Mesh, PartitionSpec, NamedSharding
from jax.experimental import mesh_utils

device_mesh = mesh_utils.create_device_mesh((4, 64))
mesh = Mesh(device_mesh, axis_names=('data', 'model'))

x = jax.random.normal(jax.random.key(0), (8192, 8192))

sharding = NamedSharding(mesh, PartitionSpec('data', 'model'))

x_sharded = jax.device_put(x, sharding)

@jax.jit
def parallel_matmul(mat):
    return mat @ mat.T

# Run
print(f"Running on {len(jax.devices())} devices...")
result = parallel_matmul(x_sharded)
print("Computation complete.")
```

Setup the Mesh (Hardware Topology)
For 256 TPUs, we might view them as a 4x64 grid. jax.devices() automatically detects all 256 chips globally.

Create Data
A large matrix: 8192 x 8192

Define Parallelism Strategy
We split the first dimension (rows) across the 'data' axis (4 ways) We split the second dimension (cols) across the 'model' axis (64 ways)

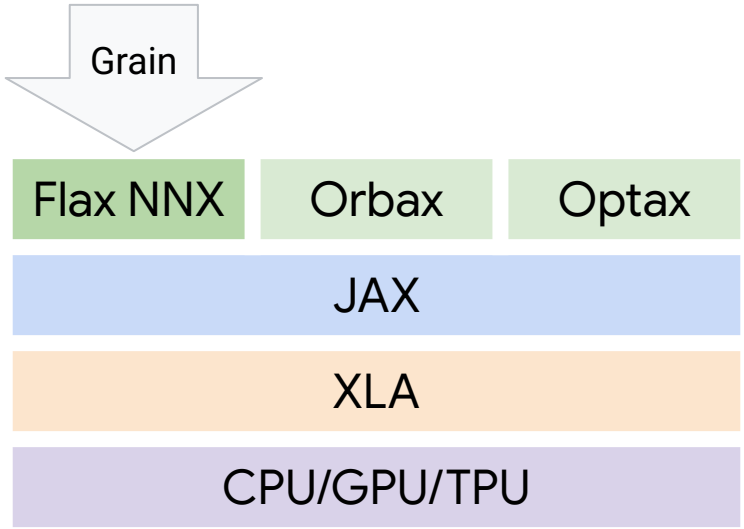
Push to Hardware
This physically scatters the matrix across the 256 chips.

Computation (JIT)
JAX sees the inputs are sharded and automatically generates the distributed communication (all-gathers, reductions) for you.

The full JAX ecosystem

JAX as a framework

- **Grain:** Data Loading (optional)
- **Flax NNX:** Neural Networks
- **Optax:** Optimizers
- **Orbax:** Checkpointing
- **JAX:** Function Transformations & NumPy API
- **XLA:** Compiler



How Google utilizes JAX today

JAX as a framework

Gemini

DeepMind

- **The Foundation of Google DeepMind**: The primary framework for advanced research.
- **Massive LLM Training**: Gemini models are trained on distributed JAX over massive TPU pods.
- **AlphaFold**: Solved the 50-year-old protein folding grand challenge (Scientific Compute).
- **MaxText**: Google's open-source, highly optimized LLM training framework written purely in Python/JAX as a platform to validate TPU & GPU performance.

Google Cloud

MaxText Github

How do customers build clusters in industry?

Orchestration

(The not so great way, but fast) Bash scripts / CLI utilities using cloud provider APIs to build clusters, networks, other infrastructure. We'll show examples shortly

(The better way) Using IaC frameworks like Terraform to build infrastructure as code so we have a deterministic way to tear down and rebuild infrastructure. Mostly Kubernetes / Slurm environments.

Here's a Google repo that customers build off of to build large scale ML clusters: https://github.com/GoogleCloudPlatform/cluster-toolkit

Commonalities between GPU and TPU: Kubernetes

Orchestration

- **Toil of raw hardware**: Managing 100s of bare-metal nodes via SSH is fragile and unscalable.
- **Kubernetes**: A unified control plane that schedules containers, networks them, and handles hardware failures automatically.
- **Device Plugins**: Expose raw accelerators (like GPUs or TPUs) as schedulable K8s resources.
- **Interaction:** We interact with and deploy our jobs mostly through YAML and k8s cli

Kubernetes handles...

Scheduling:
Decide what pods to run on which nodes

Lifecycle and health:
Keep my containers running despite failures

Scaling:
Make sets of containers bigger or smaller

Naming and discovery:
Find where my containers are now

Load balancing:
Distribute traffic across a set of containers

Storage volumes:
Provide data to containers

Logging and monitoring:
Track what's happening with my containers

Debugging and introspection:
Enter or attach to containers

Identity and authorization:
Control who can do things to my containers

XPK CLI for quick deployments on k8s

Orchestration

XPK

Tutorial that goes through a TPU cluster deployment manually can be found **here**

XPK can deploy GPU and TPU clusters, here is a TPU example

```
xpk cluster create --cluster=${CLUSTER_NAME} \
    --cluster-cpu-machine-type=n1-standard-8 \
    --num-slices=${NUM_SLICES} \
    --tpu-type=${ACCELERATOR_TYPE} \
    --zone=${ZONE} \
    --project=${PROJECT_ID} \
    --on-demand
```

30 mins

```
xpk workload create \
    --cluster ${CLUSTER_NAME} \
    --base-docker-image maxtext_base_image \
    --workload maxtext-1b-$(date +%H%M) \
    --tpu-type=${ACCELERATOR_TYPE} \
    --zone ${ZONE} \
    --project ${PROJECT_ID} \
    --command "${COMMAND}"
```

These two commands would work for 1000s of chips!

Google Cloud

TPU has an even quicker way - can run without k8s

Orchestration

```
export ACCELERATOR_TYPE="v6e-256"

gcloud alpha compute tpus queued-resources create $QR_ID \
  --project=$PROJECT_ID \
  --zone=$ZONE \
  --accelerator-type=$ACCELERATOR_TYPE \
  --runtime-version=$RUNTIME_VERSION \
  --node-id=$NODE_ID \
  --provisioning-model=FLEX-START \
  --max-run-duration=3h
```

One quick CLI and now I have 256 TPU chips on the same ICI domain

```
export VERIFY_CMD='python3 -c "import jax; print(f\"Total Devices: {jax.device_count()}, Local Devices: {jax.local_device_count()}\")"'

# Execute on all workers
gcloud alpha compute tpus queued-resources ssh $QR_ID \
  --project=$PROJECT_ID \
  --zone=$ZONE \
  --worker=all \
  --node=all \
  --command="$VERIFY_CMD"
```

One more and I just ran a Jax job on all 256 chips

Google Cloud

Anatomy of a TPU job (Condensed)

Orchestration

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: tpu-job
spec:
  parallelism: 4
  template:
    spec:
      nodeSelector:
        # Node selector to target TPU v4 slice
        cloud.google.com/gke-tpu-accelerator: tpu-v4-podslice nodes
        # Specifies the physical topology for the TPU slice.
        cloud.google.com/gke-tpu-topology: 2x2x4
      containers:
      - name: tpu-job
        command:
        - bash
        - -c
        - |
          python -c 'import jax; print("TPU cores:", jax.device_count())'
        resources:
          requests:
            # Request 4 TPU chips for this workload.
            google.com/tpu: 4
```

Key note: Each node has 4 chips (compared to usually 8 for GPUs). Our topology is 2x2x4 = 16 / 4 = parallelism of 4. We request 4 chips per node and we select nodes that map to our topology. Minor differences, the UX is pretty much the same as GPU

Everything else is the same as GPU jobs but we can omit some things like network interfaces

Google Cloud

Jax Hello World

Industry

Two dimension for measuring efficiency

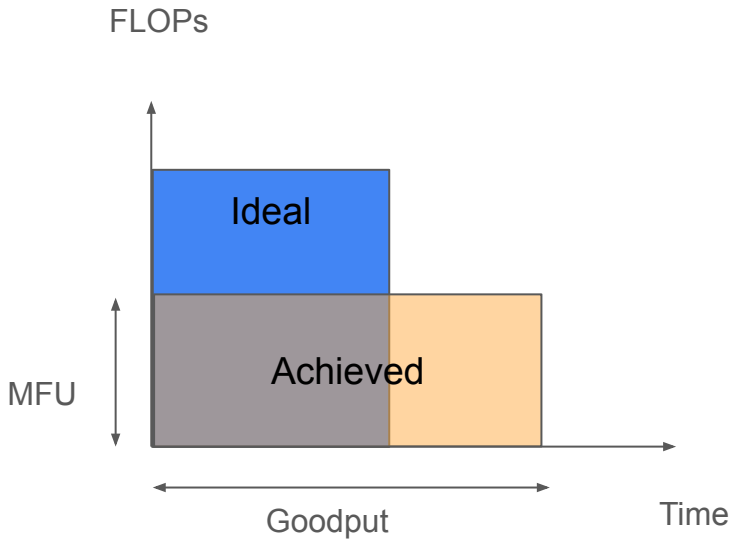
Industry examples

Model Flops Utilization (MFU) = $\frac{\text{Observed FLOPS throughput}}{\text{Peak FLOPs}}$

- Efficiency of the training regime
- Focus to optimize chips utilization of a single step. Techniques like compute communication overlap, offload of compute to host and sparse core, etc.
- SOTA ~35% in FP8 and ~45% in BF16 for training

ML Goodput = $\frac{\text{Productive time}}{\text{Total time}}$

- For a given training regime measure the efficiency of chips utilization in time of a job
- See blog for a more in depth discussion



Key themes day to day

Industry Examples

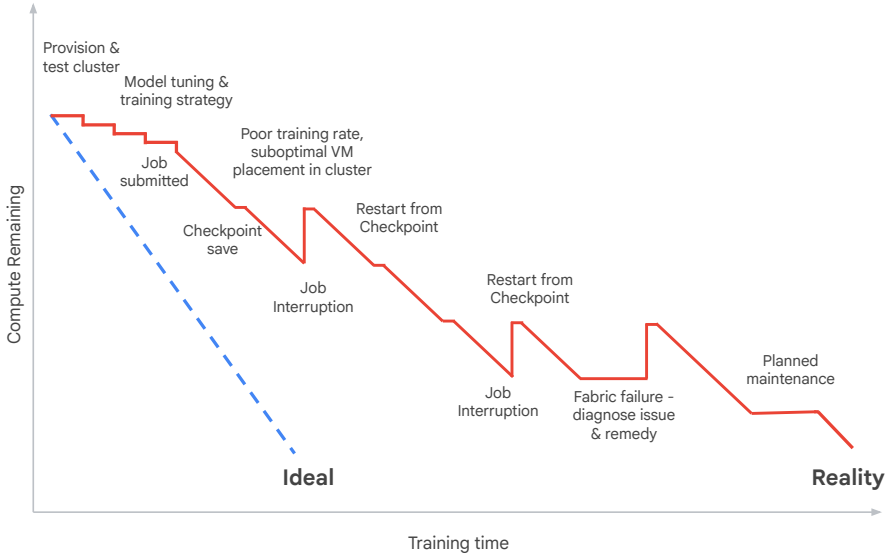
- Optimization (Many different types) → Improve MFU
- Handling hardware failures & degradation → Improve goodput
- Improving the architecture of a solution → Improve goodput
- Proof of concepts, demos, hardware validation → More “salesy”
- Capacity allocations, pricing → More “business ops”
- Reusable asset creation → Improve customer adoption

Industry reference - Hardware failures

Industry examples

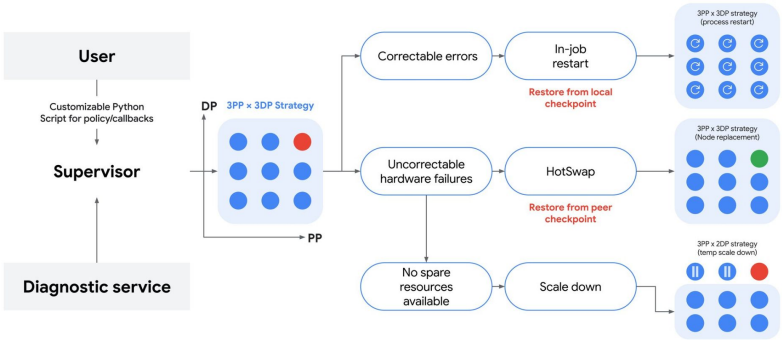
Reality: Hardware failure at scale is inevitable, degradation even more so

Reminder to read about goodput here



What do we do about it?

- Straggler detection (Proactive monitoring)
- Elastic training For in flight recovery
- Solid checkpointing strategy to recover fast
- Compilation caches
- Fast container startup
- Operational → hardware holdbacks, hot swaps



Industry reference - Sources of “Badput” - Interruptions

Industry examples

Infrastructure Recovery

The window of time after the workload is disrupted but before it can even **begin** to restart.

Re-Initialization

The **full cost** of startup (Accel Init, Training Prep, etc.) that must be paid **again** after every restart.

Wasted Progress

Time spent on training steps that are lost after a disruption, forcing a restart from the last checkpoint.

Industry reference - Meta Llama 405b

Industry examples

MFU

GPUs	TP	CP	PP	DP	Seq. Len.	Batch size/DP	Tokens/Batch	TFLOPs/GPU	BF16 MFU
8,192	8	1	16	64	8,192	32	16M	430	43%
16,384	8	1	16	128	8,192	16	16M	400	41%
16,384	8	16	16	4	131,072	16	16M	380	38%

Achieved Goodput: 90%

Training time: 54 days

Number of interruptions: 466

- 47 planned maintenance
- 419 unplanned interruptions

Source: https://arxiv.org/pdf/2407.21783

Industry reference - Google TPU Training

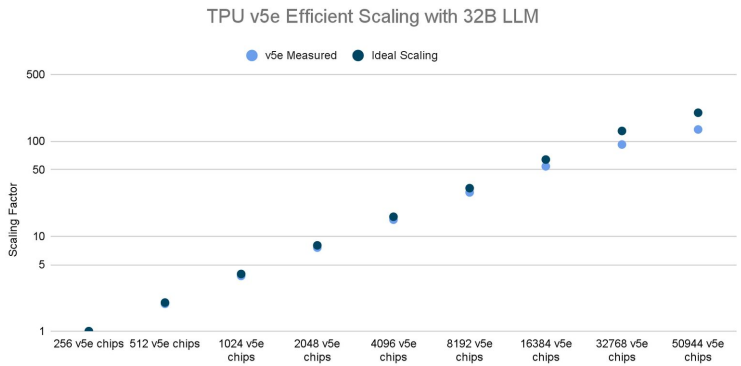
Industry examples

Worlds largest training job (2023)

TL;DR: 50,900 TPU v5e chips across 199 pods

MaxText LLM Training Results

BF16/INT8 Training	parameters	TPU v5e pods	TPU v5e chips	Observed Perf/chip	Total observed Perf	EMFU
BF16	16B	1	256	120 TFLOP/s	0.03 exa-FLOP/s	61.10%
BF16	16B	16	4096	111 TFLOP/s	0.46 exa-FLOP/s	56.56%
BF16	32B	1	256	132 TFLOP/s	0.03 exa-FLOP/s	66.86%
BF16	32B	16	4096	123 TFLOP/s	0.50 exa-FLOP/s	62.26%
BF16	64B	1	256	118 TFLOP/s	0.03 exa-FLOP/s	59.90%
BF16	64B	16	4096	105 TFLOP/s	0.43 exa-FLOP/s	53.29%
BF16	128B	1	256	110 TFLOP/s	0.03 exa-FLOP/s	56.06%
BF16	128B	16	4096	100 TFLOP/s	0.41 exa-FLOP/s	50.86%
BF16	32B	199	50944	88 TFLOP/s	4.48 exa-FLOP/s	44.67%
INT8 Quant	**32B**	**199**	**50944**	**104.4 TOP/s**	**5.32 exa-OP/s**	**52.99%**



From the field

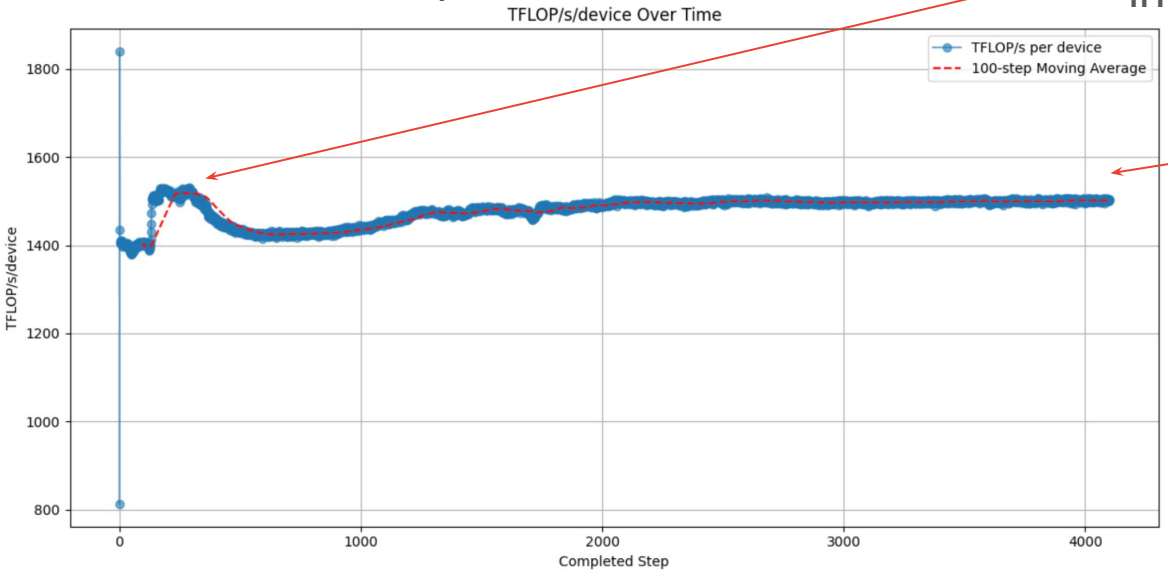
Industry reference - Hardware & Training validation

Industry examples

Customer ask → Jax on CUDA validation, want to see a 24h run and measure MFU for a scaled training job. Want to see full networking stack for RDMA deployed

B200 & MaxText recipe: found here

Interesting investigation



33% MFU

This was a 24h run to pre-train Llama3 70b in FP8 on 256 B200 GPUs

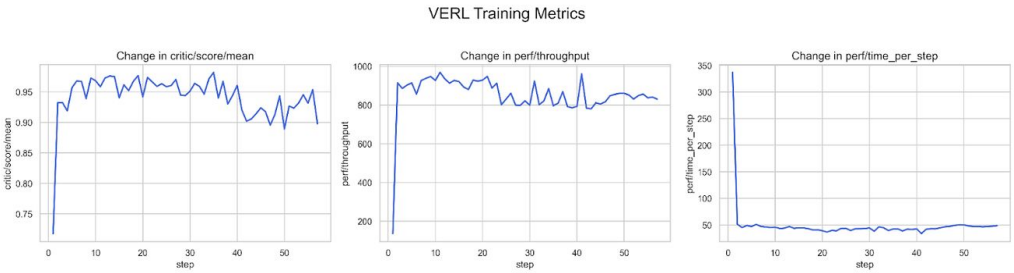
Industry reference - PoCs / Solution Ach. / Asset creation

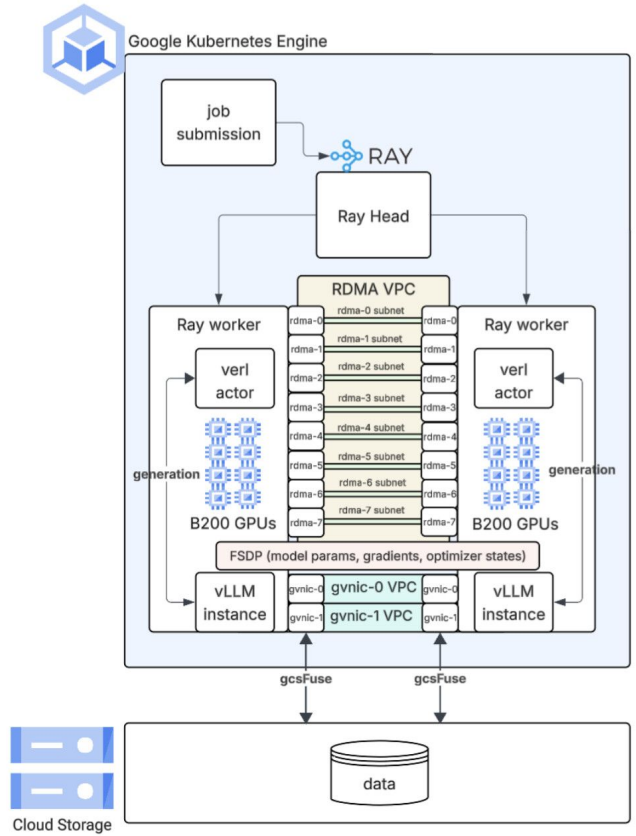
Industry examples

Customer ask → Key themes → "*I want to do RL, how would I do this on GCP?*" → "*I want OSS, not proprietary solutions*"

Recipe / architecture doesn't exist? Let's make one & ship to customers→ [Scaling RL on GKE with VERL]

Fairly SOTA MFU → 33% for RL





Industry reference - Optimization

Industry examples

Key themes →

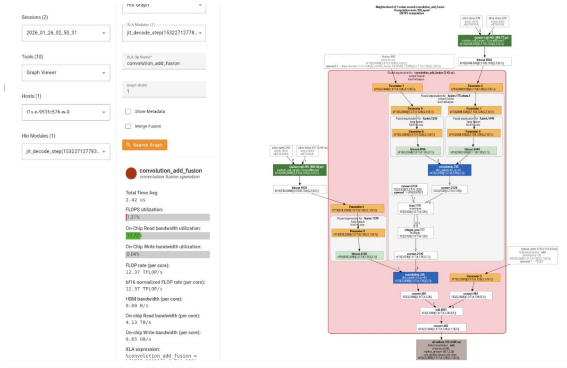
"*My training job is slow, my MFU is low*"
"My inference latency is bad, p99 latency suffers"
"Could I achieve a better perf / cost on chip X, Y, Z?
"What type of storage should I use?"
"I heard TPUs are cheap and performant, what about those?"
Ambiguous asks → "*Am I optimized????*"

Will share some "simplified" views of some problems customers face.

Training is the most complex (lot of factors at play because of distributed nature, inference not as much because it's mostly single host)

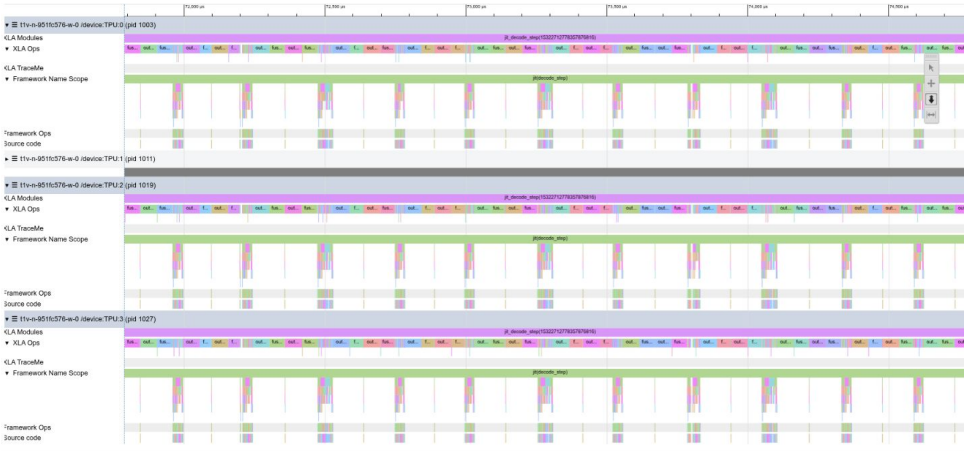
Industry reference - Optimization - Slow jobs

Industry examples



First step → Get me a profile

GPUs → NVIDIA Nsights Systems, PyTorch
TPUs → Xprof



Second step → Analyze traces, graphs for bottlenecks

Jax profiling guide here

Note: profiling is a skill all in of itself

Industry reference - Optimization - Inference perf cost

Industry examples

Customer ask →

Billions of offline batch inference requests
Using B200 today, can't get more capacity in a geo-loc
Model is gpt-oss-120b
Avg. input token 10K, output token 200, sys prompt 1500
Only care about performance cost for throughput

Let's benchmark (vLLM) →

vllm serve openai/gpt-oss-120b --tensor-parallel-size $TP
--enable-prefix-caching --max-model-len $MAX_MODEL_LEN

```
vllm bench serve \
    --backend vllm \
    --model openai/gpt-oss-120b \
    --dataset-name random \
    --num-prompts 1000 \
    --random-prefix-len=1500 \
    --random-input-len=10000 \
    --random-output-len=200
```

Rtx pro 6000 x 2

```
Successful requests:                     983
Failed requests:                         17
Benchmark duration (s):                  490.02
Total input tokens:                      9830000
Total generated tokens:                  442350
Request throughput (req/s):              2.01
Output token throughput (tok/s):         902.72
Peak output token throughput (tok/s):    4272.00
Peak concurrent requests:                983.00
Total token throughput (tok/s):          20963.19
----------Time to First Token----------
Mean TTFT (ms):                          232961.54
Median TTFT (ms):                        229715.83
P99 TTFT (ms):                           466014.88
-----Time per Output Token (excl. 1st token)------
Mean TPOT (ms):                          252.38
Median TPOT (ms):                        283.34
P99 TPOT (ms):                           286.38
----------Inter-token Latency----------
Mean ITL (ms):                           252.38
Median ITL (ms):                         355.59
P99 ITL (ms):                            375.55
```

B200 x 1

```
Successful requests:                     983
Failed requests:                         17
Benchmark duration (s):                  630.69
Total input tokens:                      9830000
Total generated tokens:                  442350
Request throughput (req/s):              1.56
Output token throughput (tok/s):         701.38
Peak output token throughput (tok/s):    4000.00
Peak concurrent requests:                983.00
Total token throughput (tok/s):          16287.57
----------Time to First Token----------
Mean TTFT (ms):                          300630.59
Median TTFT (ms):                        294766.84
P99 TTFT (ms):                           597866.07
-----Time per Output Token (excl. 1st token)------
Mean TPOT (ms):                          305.85
Median TPOT (ms):                        340.14
P99 TPOT (ms):                           342.62
----------Inter-token Latency----------
Mean ITL (ms):                           305.87
Median ITL (ms):                         457.67
P99 ITL (ms):                            472.65
```

Without sharing pricing details, perf / cost on RTX Pro pulled ahead

Solution → RTX Pro 6000

Why? → Still Blackwell (Need the native FP4 support to help with gpt-oss-120b perf) but a cheaper and more available chip → not a lot of decode, so the loss of mem bandwidth won't hurt as much.

Industry reference - Optimization - 1P to 3P

Industry examples

Customer ask →
Billions of ***online*** inference requests
Need E2E latency as low as possible
Small requests, input → 350, output → 12

Gemini flash lite →

Model	P50
Gemini Flash Lite	704ms

Gemini is a large model, 1P models have some overhead because of tenancy and other factors.

Solution →
Fine tune a medium MoE model like gpt-oss-120b, self serve on GPUs for fine grained serving optimizations compared to 1P

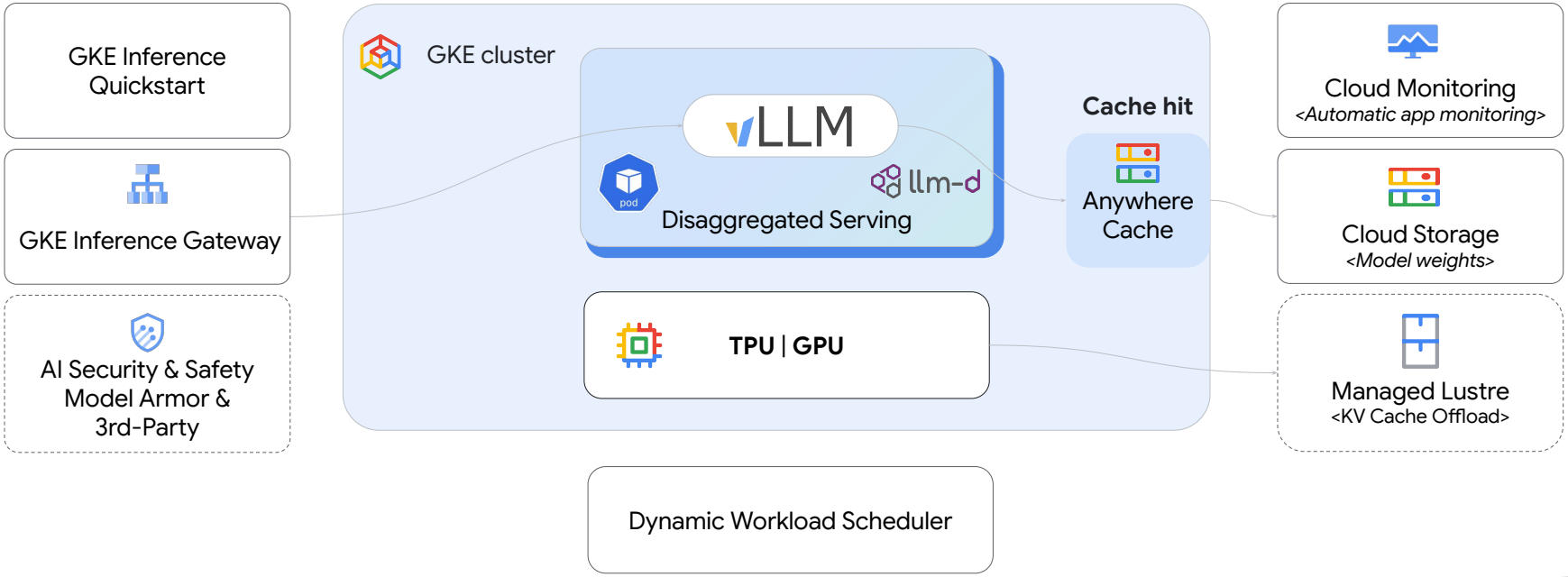
Rinse and Repeat our vLLM commands from before

```
============ Serving Benchmark Result ============
Request throughput (req/s):          194.98
Output token throughput (tok/s):     2339.76
Peak output token throughput (tok/s): 2532.00
Peak concurrent requests:            245.00
Total token throughput (tok/s):      64733.45
----------Time to First Token----------
Mean TTFT (ms):                      42.42
Median TTFT (ms):                    42.32
P99 TTFT (ms):                       56.19
-----Time per Output Token (excl. 1st token)------
Mean TPOT (ms):                      13.64
Median TPOT (ms):                    13.78
P99 TPOT (ms):                       15.70
----------Inter-token Latency----------
Mean ITL (ms):                       13.64
Median ITL (ms):                     13.67
P99 ITL (ms):                        19.70
==================================================
```

New p99 E2E: 244ms

Industry reference - Inference Sol'n Arch. on GCP

Industry examples



Industry reference - Optimization - Storage

Industry examples

Customer ask →

Large dataset (>1PB), stored on Google Cloud Storage
Image data, all small files 100kb
Using GCS Fuse ← This is a fairly common pattern and usually very high performance
OSS reference model (ViT)
Low MFU, profiles show bottleneck on storage IO

Solution →

Storage re-architecture, best practices outlined here
Migrate data loading code to use Dataflux for GCS
Dataflux is a 1P storage framework for PyTorch, has some key performance optimizations like image composing

File size / count	Tool	Training time (s)
100 KiB / 500000 files	Direct GCS API calls	1,299
	Connector Map-style Dataset	515
500 KiB / 2.2m files	Direct GCS API calls	6,499
	Connector Map-style Dataset	2,058
3 MiB / 50000 files	Direct GCS API calls	399
	Connector Map-style Dataset	277
150 MiB / 5000 files	Direct GCS API calls	1,396
	Connector Map-style Dataset	1,173

Source: https://github.com/GoogleCloudPlatform/gcs-connector-for-pytorch/tree/main

Able to achieve ~250ms latency between infrastructure and GCS for a 32mb batch of images, GPU step time is >250ms so storage bottleneck won't be a factor

Q&A

Erik Saarenvirta

W: esaarenvirta@google.com

P: erik.saarenvirta93@gmail.com

L: www.linkedin.com/in/erik-saarenvirta/