Hello

Introduction to TPU & Jax

10 / 07

Outline

TPU

What is TPU

How to use TPU

Why TPU

Jax

What is Jax

How to use Jax

Why Jax

Google Cloud & TPU

an Introduction

Experts

Google Developers

What is TPU

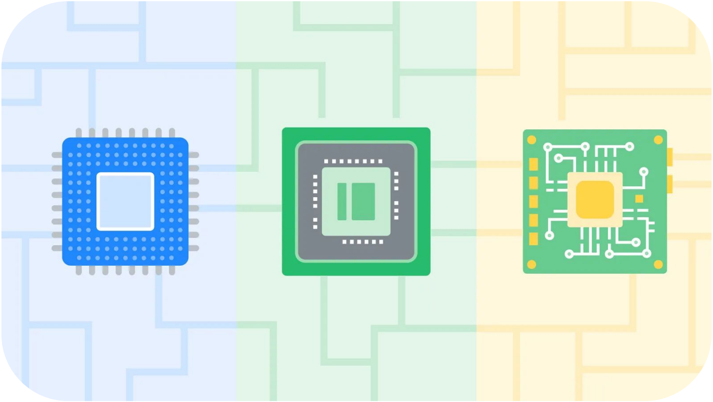
*A **TPU** (Tensor Processing Unit) is a special kind of computer chip made by Google, specifically for machine learning (ML) and deep learning.*

CPU, GPU, and TPU

CPU → general-purpose chips that can handle a diverse range of tasks.

GPU → accelerated compute tasks, from graphic rendering to AI workloads.

TPU → designed from the ground up to run AI-based compute tasks

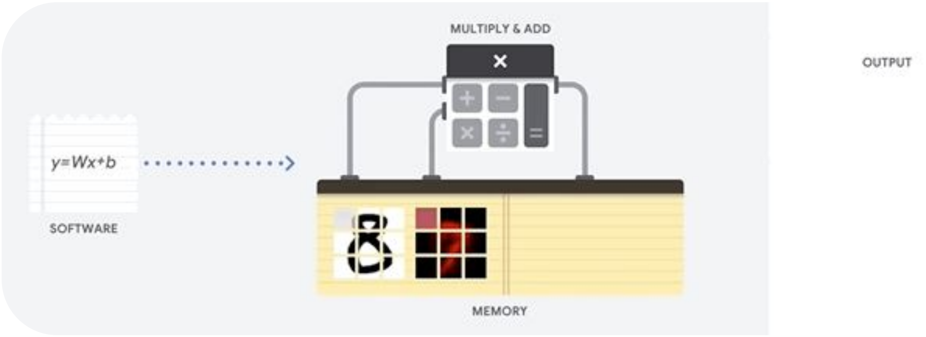


What is CPU?

A *general-purpose* processor based on the *von Neumann architecture*.

CPU loads values from memory, performs calculation and stores the result back

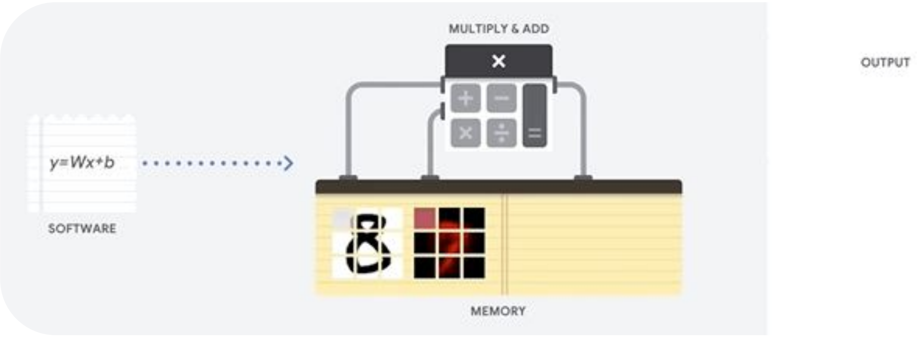
Memory access is slow compared calculation speed and limit total throughput



What is CPU? **Flexibility**

You can load any kind of software on a CPU

- word processing on a PC
- controlling rocket engines
- executing bank transactions
- classifying images with a neural network.



What is GPU?

Contain thousands of Arithmetic Logic Units (ALUs) in a single processor

- (modern GPU between 2,500–5,000 ALUs).

This means you can execute thousands of multiplications and additions **simultaneously**.

What is GPU?

This GPU architecture works well on applications with **massive parallelism**

- Like a matrix operations in a neural network

What is GPU?

Still, GPU is a *general-purpose* processor that has to support different applications.

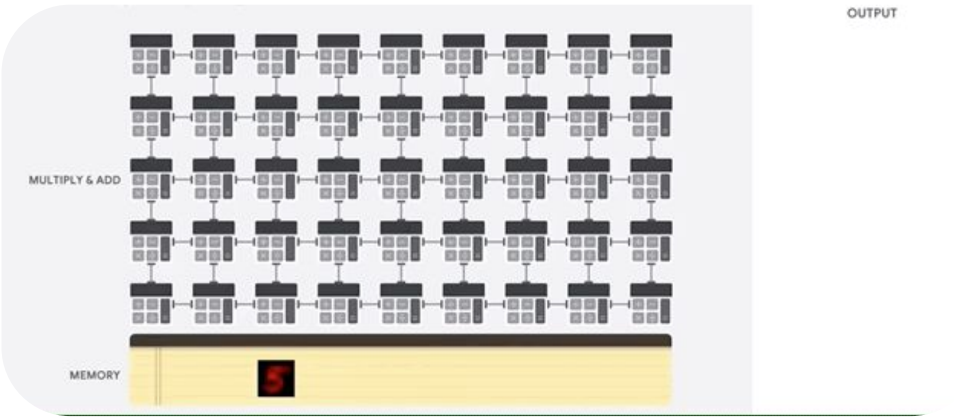
Same problem as CPUs, for every calculation

- **GPU access shared memory to read operands & store intermediate calculation.**

What is TPU?

Google designed TPUs as a **matrix processor** specialized for **neural network**.

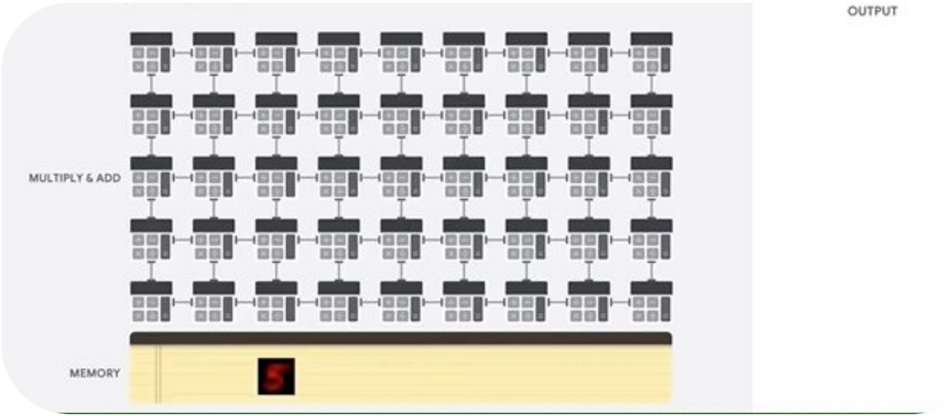
- Not general-purpose
- Handle massive matrix operations used in neural networks at fast speeds



What is TPU?

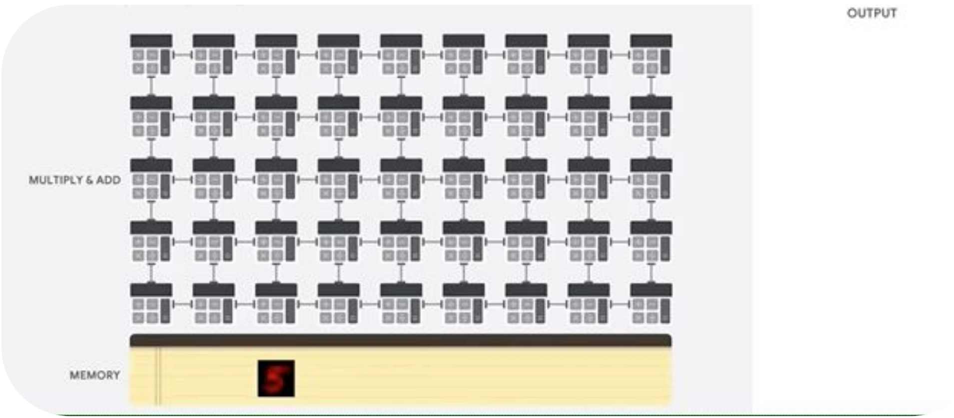
Designed for **matrix – a combination of multiply and accumulate operations**.

- thousands of multiply-accumulators that are directly connected to each other to form a large physical matrix, a systolic array architecture.



What is TPU?

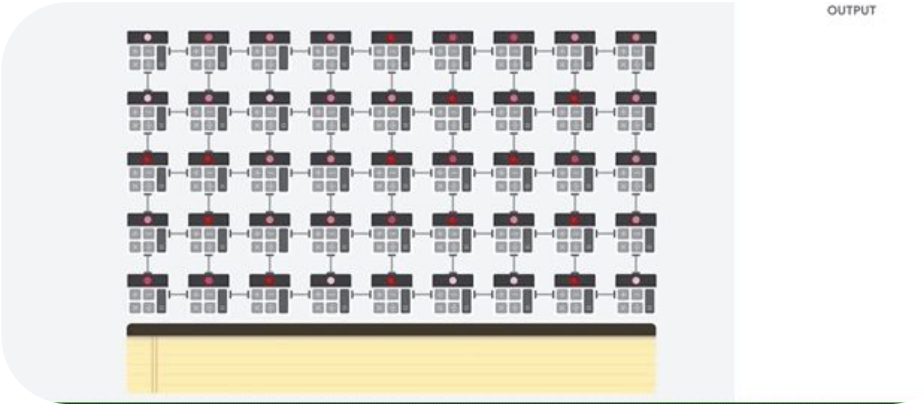
- TPU host streams data into an infeed queue.
- TPU loads data from the infeed queue and stores them in HBM memory.
- When the computation completed, TPU loads the results into the outfeed queue.
- TPU host then reads the results from the outfeed queue, stores in the host's memory



What is TPU?

1. TPU loads data from HBM memory.
2. Each multiplication is executed, passed to the next multiply-accumulator.
3. Output is summation of all results between the data and parameters.

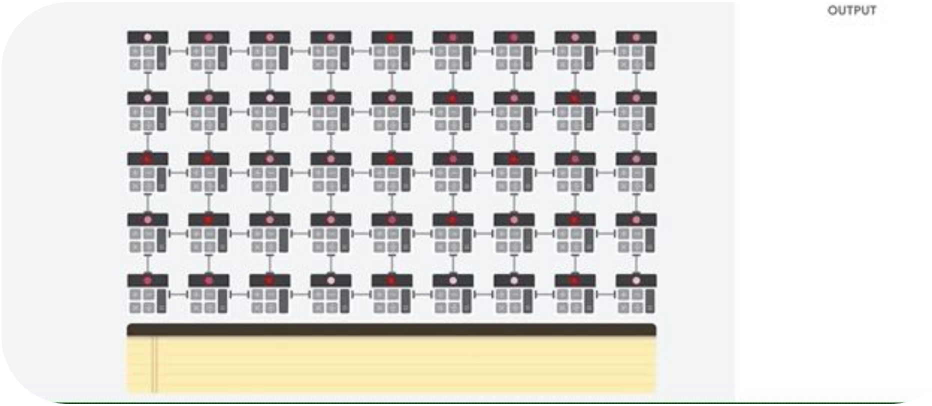
No memory access is required during the matrix multiplication process.



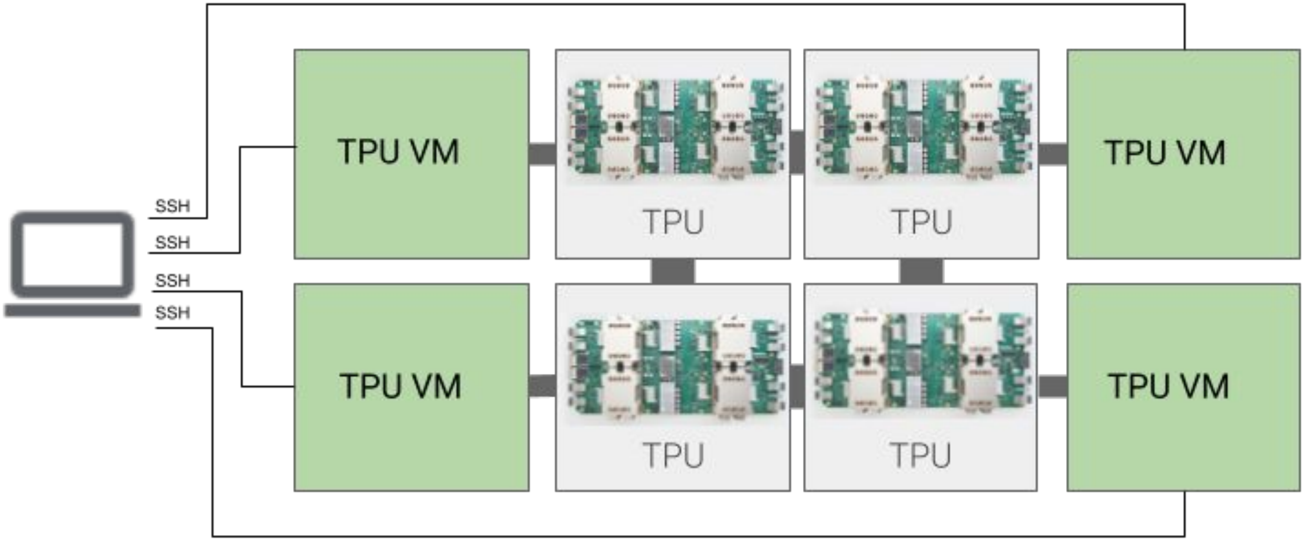
What is TPU?

As a result,

TPUs can achieve a high-computational throughput on neural network calculations.



TPU Architecture



TPU Architecture

type	*functionality*	*comparison to GPU Machine*
TPU Chip	the physical accelerator (matrix units + HBM)	*one GPU*
TPU VM	Linux VM attached to your chips to run code.	*GPU Machine*
Slice	a reserved set of chips you train on as one job (e.g., v4-64).	*N-GPU allocation*
TPU Pod	many chips wired as one big cluster.	*GPU superpod/cluster*
TPU	the overall platform (hardware + interconnect + software).	*GPU platform*

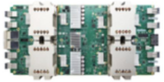
The TPU Chip

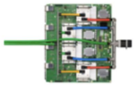
What type of chip do we have?

The TPU Chip

What type of chip do we have?

2015	2018	2020	2022	2023	2024
v1	***v2***	***v3***	***v4***	***V5e, v5p***	***V6e***



The TPU Chip

generation	*year*	*peak compute (bf16)*	*memory*	*relative performance*
TPU v2	2018	~45 TFLOP	8GB	first to support training (bfloat16)
TPU v3	2020	~123 TFLOPs	32 GB	~2–3 × in v2 performance
TPU v4	2022	~275 TFLOPs	32 GB	~2.1× in v3 performance
TPU v5e	2023	~197 TFLOPs	16 GB HBM	~2.7× *performance per dollar* vs v4
TPU v6e	2024	~920 TFLOPS	32 GB HBM	~4.7 × in v5e performance

Benchmarking

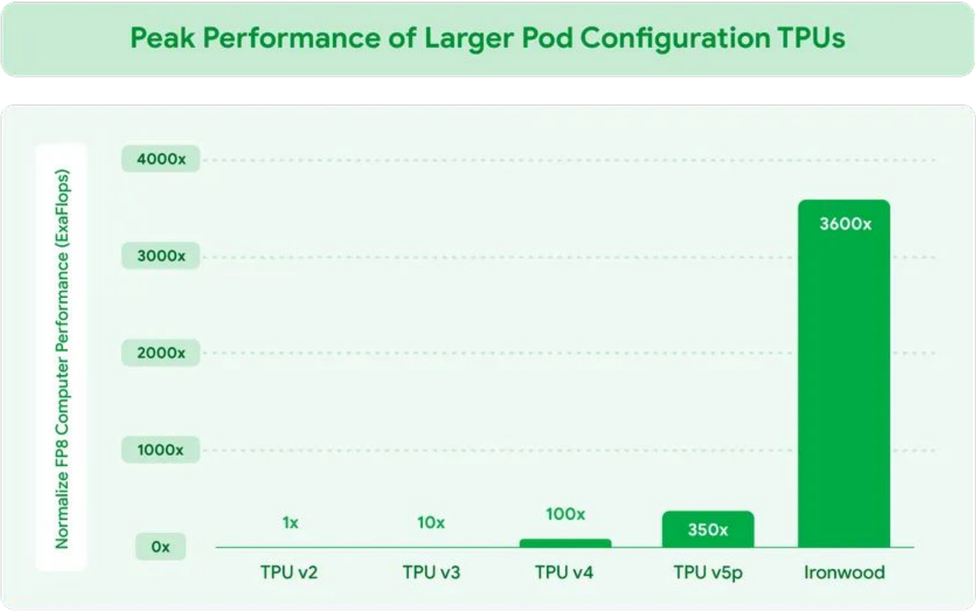
rank	*chip*	*peak compute*	*memory (GB)*	*Relative compute*
1 🥇	**B100**	~1,800 TFLOPs	192	~655 %
2 🥈	**TPU v6e**	~920 TFLOPs	32	~295 %
3 🥉	**H100**	~990 TFLOPs	80	~317 %
4	**A100**	~312 TFLOPs	40	**100 % (baseline)**
5	**TPU v4**	~275 TFLOPs	32	~88 %
6	**TPU v5e**	~197 TFLOPs	16	~63 %
7	**L40**	~182 TFLOPs	48	~59 %
8	**V100**	~125 TFLOPs	16	~40 %
9	**TPU v3**	~123 TFLOPs	16	~39 %
10	**TPU v2**	~45 TFLOPs	8	~14 %

The TPU Chip

Expecting anything new?

Yes!

The TPU Chip



Peak Performance of Larger Pod Configuration TPUs
Normalize FP8 Computer Performance (ExaFlops)
4000x
3000x
2000x
1000x
0x
1x
10x
100x
350x
3600x
TPU v2
TPU v3
TPU v4
TPU v5p
Ironwood

Apply for TPU

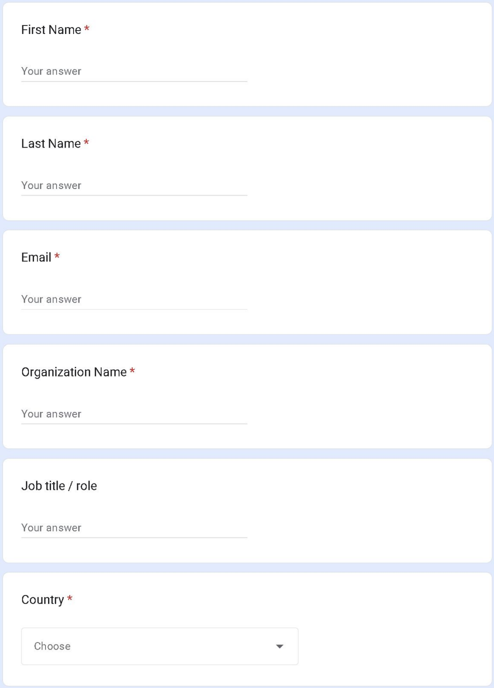
For people outside of Google (us),

the only way to access TPUs for free is through the

“TPU Research Cloud”

Apply for TPU

- Resource one can typically get within a week after applying
- more resources available upon request


First Name *

Your answer

Last Name *

Your answer

Email *

Your answer

Organization Name *

Your answer

Job title / role

Your answer

Country *

Choose

the application form is extremely simple

TPU Quota

- ***50*** *spot Cloud TPU v2-8 device(s) in zone us-central1-f*
- ***50*** *spot Cloud TPU v3-8 device(s) in zone europe-west4-a*
- ***32*** *on-demand TPU v4 chips in zone us-central2-b*
- ***32*** *spot Cloud TPU v4 chips in zone us-central2-b*

typical TPU quota for the average user

- ***256*** *spot Cloud TPU v4 chips in zone us-central2-b*
- ***32*** *on-demand TPU v4 chips in zone us-central2-b*
- ***64*** *spot Cloud TPU v5e chips in zone us-central1-a*
- ***64*** *spot Cloud TPU v5e chips in zone europe-west4-b*
- ***64*** *spot Cloud TPU v6e chips in zone europe-west4-a*
- ***64*** *spot Cloud TPU v6e chips in zone us-east1-d*

current quota of our lab

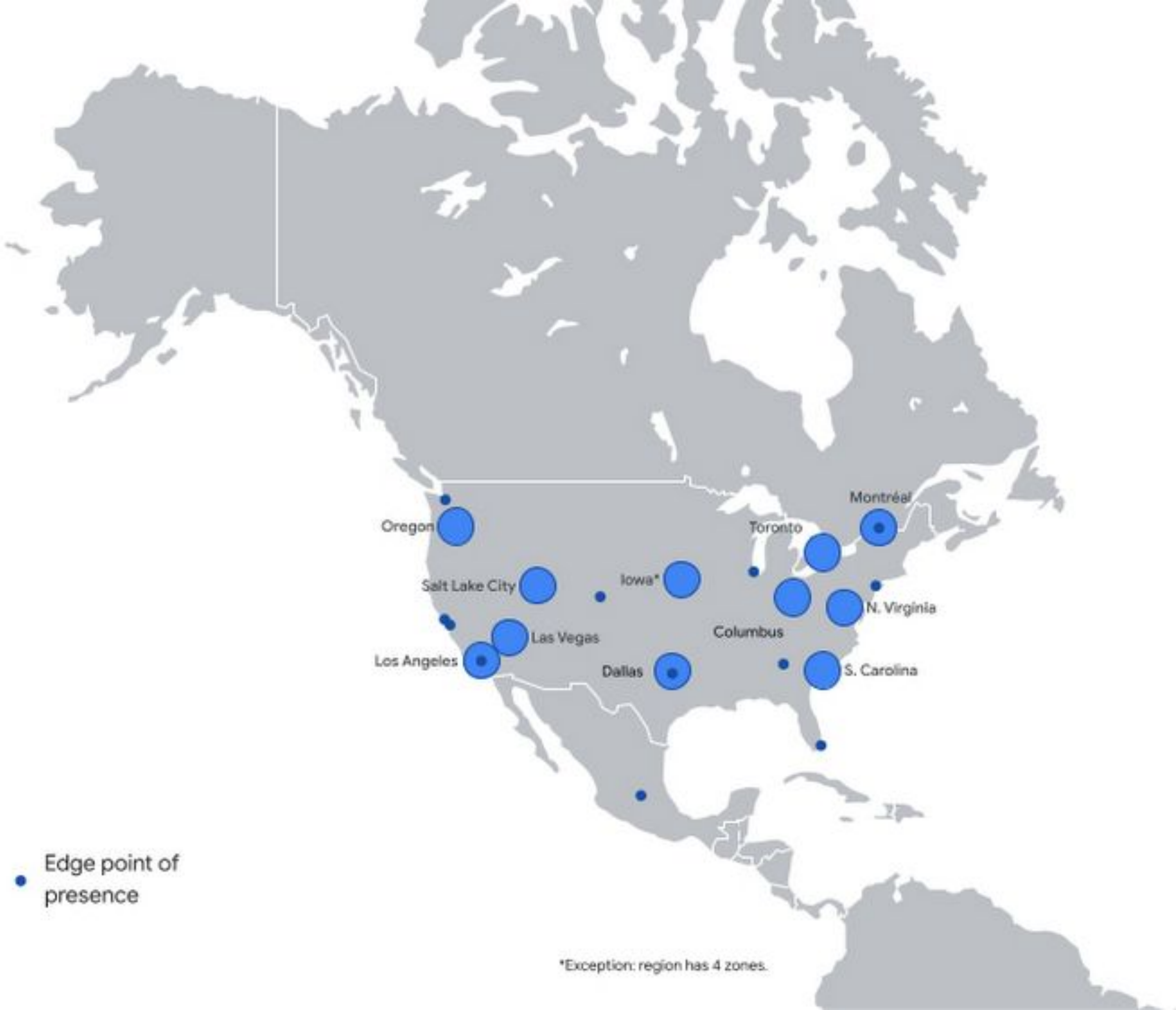
TPU Zones

- Google cloud separate their resource into different zones
    - TPUs
    - Storage

Google Cloud Platform

North America

11	34	28
Regions	Zones	PoPs



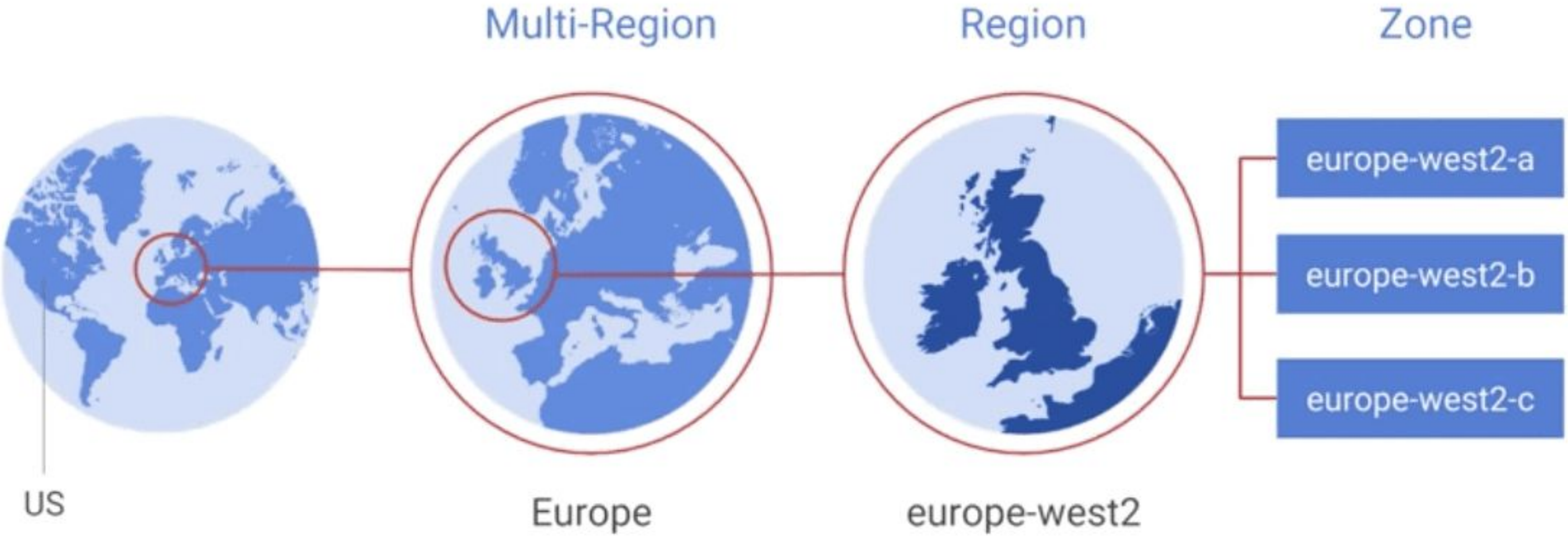
Current region with 3 zones

Future region with 3 zones

Edge point of presence

*Exception: region has 4 zones.

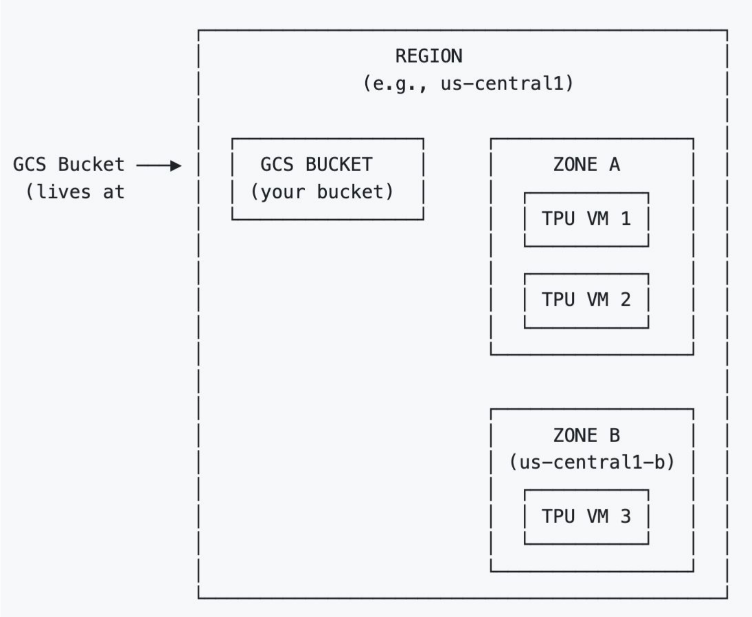
Google Cloud Platform is organized into regions and zones



TPU Zones

Same region is fast and cheap, cross-Region incurs costs

TPU VM Framework



TPU Queuing

Two types: <u>on-demand</u> & <u>spot</u>

- V4-128, availability is very unstable, sometimes, especially weekends, hard to get
- V5, okay, easy to use, but random error
- V6, almost completely unusable

TPU v5 issues

Our observation:

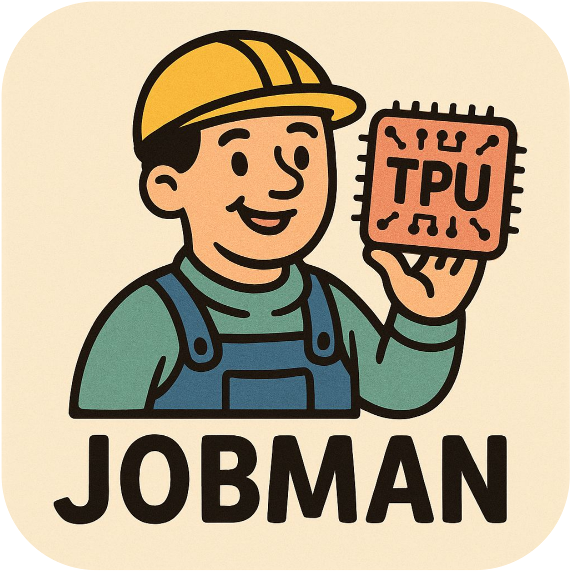
- Some v5 TPU give random NaN loss during training

Cannot —exclude unusable nodes

Jobman

A toolbox

- Submit job with config
- Auto setup for storage & environment
- Check job status
- Re-queue after preempt



https://github.com/Zephyr271828/jobman

Thanks to @Yufeng Xu for creating the toolbox

Jobman

Can be host on any Princeton cluster

Basic Commands

Purpose	Command
Create a new job	jobman create <config_path>
Check all jobs status	jobman list
Resume an existing job	jobman resume <job_id>
Kill the backend process for a job	jobman cancel <job_id>
Kill process and delete TPU resources	jobman delete <job_id>
Kill, delete, and clean logs	jobman clean <job_id>

```
(tpu) [tl0463@della-vis1 language]$ jobman list
|   Job ID | User    | Name                                   | Accelerator     | Zone            | Host0 IP           | Status     |
|----------|---------|----------------------------------------|-----------------|-----------------|--------------------|------------|
|   000002 | tl0463  | pretrain_llama3.1_1b_finewebedu_000002 | v4-128          | us-central2-b   | 35.186.71.232      | DEAD       |
|   000003 | tl0463  | pretrain_llama3.1_1b_finewebedu_000003 | v4-128          | us-central2-b   | 35.186.114.4       | DEAD       |
|   000004 | tl0463  | pretrain_llama3.1_1b_finewebedu_000004 | v4-128          | us-central2-b   | 35.186.88.64       | DEAD       |
|   000005 | tl0463  | pretrain_llama3.1_1b_finewebedu_000005 | v4-128          | us-central2-b   | 35.186.0.98        | DEAD       |
```

https://github.com/Zephyr271828/jobman

Thanks to @Yufeng Xu for creating the toolbox

TPU Costs

Operation cost

- Class A operations (e.g., uploading files) → $0.0050 / 1,000 operations
- Class B operations (e.g., accessing files) → $0.0004 / 1,000 operations

Data Transfer

- Not cost in same region (local to cloud)
- High cost cross region (1TB data would cost at least $20)

Compute engine

- The usage of TPUs is often covered by the TPU Research Cloud program.
- For a quick estimation, the price of a TPU v4-8 is $12.88 / hour

TPU Costs

- Reduce I/O operations from VM to storage by compressing datasets.
- Ensure your storage and VM are in the same zone.
- Do not create multi-region storage.
- The best practice is to process everything locally and upload to the storage

Google Cloud

other useful services

Cloud Storage

Google Cloud Storage is an online service that lets you store, access, and manage any amount of data securely on Google's servers from anywhere.

Google Cloud Storage

BUCKETS

Cloud Storage

Buckets are the basic containers that hold your data as objects.

Everything that you store in Cloud Storage must be contained in a bucket.

Cloud Storage

Buckets are the basic containers that hold your data as objects.

Everything that you store in Cloud Storage must be contained in a bucket.

Cloud Firestore

Google’s serverless NoSQL document database

- store JSON-like documents inside collections
- query with indexes
- get real-time listeners

Cloud Run

A fully managed platform that runs containers over HTTPS

- just docker build and deploy.

Free Credit for Cloud

Use for your non-TPU projects?

Start running workloads for free

Create an account to evaluate how Google Cloud products perform in real-world scenarios. New customers get $300 in free credits to run, test, and deploy workloads, including Google-recommended, pre-built solutions. All customers can use 20+ products for free, up to monthly usage limits.

$300

in free credits

20+

free products

Only pay for what you use

With Google Cloud's pay-as-you-go pricing structure, you only pay for the services you use. No up-front fees. No termination charges. Pricing varies by product and usage—view detailed price list.

Save up to 57% on workloads

Google Cloud saves you money over other providers through automatic savings based on monthly usage and by pre-paying for resources at discounted rates. For example, save up to 57% with committed use discounts on Compute Engine resources like machine types or GPUs.

Intro to JAX

examples and personal experiences

XLA – the compiler

Your Python code
→ PyTorch ops
→ Precompiled CUDA kernels & CUDA libraries (cuBLAS/cuDNN)
← these were built with *nvcc* ahead of time
→ CUDA driver runs it on the GPU

Your Python code
→ jax.jit / tf.function traces the math
→ *XLA* compiles it into GPU code
→ CUDA driver runs it on the GPU

XLA – the compiler

XLA is a smart **compiler** for your ML code.

When you write JAX (or TF) functions and use jit

- XLA looks at the math you're doing
- fuses lots of tiny ops into a few big ones
- picks fast kernels
- generates machine code tailored for the device you're using (GPU, TPU, or CPU)

That means less memory traffic, fewer kernel launches, and faster runs without you writing CUDA or TPU code yourself.

TPU Runtime

On NVIDIA GPU, you use CUDA

What is CUDA?

CUDA is NVIDIA's GPU platform and toolkit. It includes:

- a programming model & APIs (CUDA C/C++/Python),
- a compiler (**nvcc**) that turns your code into GPU instructions (PTX/SASS), a runtime/driver to launch kernels,
- and fast math libraries (**cuBLAS, cuDNN**, etc.).

TPU Runtime

On Google TPU, you directly use TPU Runtime

- **On NVIDIA GPU:**
  your code → (PyTorch kernels or JAX via XLA) →
  CUDA driver/libs → GPU runs it.

- **On TPU:**
  your code → **XLA compiles to TPU executable** →
  TPU runtime (PJRT + libtpu) → TPU runs it.

Introducing Jax

The language that *drives* those compilers for us: **JAX**

- JAX = NumPy-like Python + **transformations** (jit, grad, vmap, pjit)
- jit → hands a graph to **XLA** → device executable
  PJRT picks the **backend** (GPU/TPU/CPU) and runs it
- Same code, different hardware; no CUDA/TPU kernel writing

Introducing Jax

What it is:

JAX is a Python library made by Google Research
for fast math and ML on CPU, GPU, and TPU.

Goal:

Make code that looks like **NumPy**, but runs **as fast as compiled code**.

Why built:

To get both **speed** and **flexibility**
— something NumPy can't and PyTorch handles differently.

Introducing Jax

How it works:

You write normal math functions

JAX can **auto-differentiate**, **compile**, and **run them in parallel**.

Key idea:

You don’t write models step-by-step like PyTorch;

you write **pure functions** and apply transforms

(jit → compile to machine code, grad → get gradients, etc.).

Why not just PyTorch:

PyTorch is easier for everyday ML

but JAX is **better for research**, **scaling**, and **TPU use**

it’s more functional, more composable, and often faster.

Common Libraries / Functionalities in PyTorch vs. JAX

torch.Tensor	jax.numpy
torch.optim	optax
torch.nn	flax.linen
torch.utils.data	tensorflow_datasets
torch.distributed	pmap

Model Definition

PyTorch

```python
class MLP(nn.Module):
    def __init__(self, input_dim: int, hidden_sizes=(1024, 512), num_classes=10):
        super().__init__()
        self.hidden_sizes = hidden_sizes
        self.num_classes = num_classes
        layers = []
        prev_dim = input_dim
        for h in hidden_sizes:
            layers.append(nn.Linear(prev_dim, h))
            layers.append(nn.ReLU())
            prev_dim = h
        layers.append(nn.Linear(prev_dim, num_classes))
        self.net = nn.Sequential(*layers)

    def forward(self, x):
        return self.net(x)
```

JAX

```python
class MLP(nn.Module):
    hidden_sizes: Tuple[int, ...] = (1024, 512)
    num_classes: int = 10

    @nn.compact
    def __call__(self, x):
        for h in self.hidden_sizes:
            x = nn.Dense(h)(x)
            x = nn.relu(x)
        x = nn.Dense(self.num_classes)(x)
        return x
```

Model Definition

PyTorch

```python
class MLP(nn.Module):
    def __init__(self, input_dim: int, hidden_sizes=(1024, 512), num_classes=10):
        super().__init__()
        self.hidden_sizes = hidden_sizes
        self.num_classes = num_classes
        layers = []
        prev_dim = input_dim
        for h in hidden_sizes:
            layers.append(nn.Linear(prev_dim, h))
            layers.append(nn.ReLU())
            prev_dim = h
        layers.append(nn.Linear(prev_dim, num_classes))
        self.net = nn.Sequential(*layers)

    def forward(self, x):
        return self.net(x)
```

JAX

```python
class MLP(nn.Module):
    hidden_sizes: Tuple[int, ...] = (1024, 512)
    num_classes: int = 10

    @nn.compact
    def __call__(self, x):
        for h in self.hidden_sizes:
            x = nn.Dense(h)(x)
            x = nn.relu(x)
        x = nn.Dense(self.num_classes)(x)
        return x
```

Model Definition

PyTorch

```python
class MLP(nn.Module):
    def __init__(self, input_dim: int, hidden_sizes=(1024, 512), num_classes=10):
```

PyTorch builds and stores layers upfront
JAX builds them dynamically on first call

```python
            layers.append(nn.Linear(prev_dim, h))
            layers.append(nn.ReLU())
            prev_dim = h
        layers.append(nn.Linear(prev_dim, num_classes))
        self.net = nn.Sequential(*layers)

    def forward(self, x):
        return self.net(x)
```

JAX

```python
class MLP(nn.Module):
```

```python
    def __call__(self, x):
        for h in self.hidden_sizes:
            x = nn.Dense(h)(x)
            x = nn.relu(x)
        x = nn.Dense(self.num_classes)(x)
        return x
```

Model Initialization

PyTorch

```
model = MLP(28 * 28)
```

JAX

```
rng = jax.random.PRNGKey(args.seed)

x = jnp.zeros((1, 28 * 28), jnp.float32)

params = model.init({"params": rng},
x)["params"]
```

Model Initialization

PyTorch

```
model = MLP(28 * 28)
```

JAX

```
rng = jax.random.PRNGKey(args.seed)

x = jnp.zeros((1, 28 * 28), jnp.float32)

params = model.init({"params": rng},
x)["params"]
```

Model Initialization

PyTorch

```
model = MLP(28 * 28)
```

JAX

```
rng = jax.random.PRNGKey(args.seed)
```

PyTorch uses global RNG
JAX needs explicit PRNG key

Model Initialization

PyTorch

```
model = MLP(28 * 28
```

JAX

```
rng = jax.random.PRNGKey(args.seed)

x = jnp.zeros((1, 28 * 28), jnp.float32)

params = model.init({"params": rng},
x)["params"]
```

Model Initialization

PyTorch

```
model = MLP(28 * 28
```

JAX

```
rng = jax.random.PRNGKey(args.seed)
```

PyTorch uses explicit model configuration

JAX infers model configuration from example input

Model Initialization

PyTorch

```
model = MLP(28 * 28
```

JAX

```
rng = jax.random.PRNGKey(args.seed)

x = jnp.zeros((1, 28 * 28), jnp.float32)

params = model.init({"params": rng},
x)["params"]
```

Model Initialization

PyTorch	JAX
`model = MLP(28 * 28`	`rng = jax.random.PRNGKey(args.seed)`

PyTorch stores weights in model
JAX stores them externally

Model Training

JAX

```
tx = optax.adamw(args.learning_rate)
opt_state_cpu = tx.init(params_cpu)
params_repl = flax.jax_utils.replicate(params_cpu)
opt_state_repl = flax.jax_utils.replicate(opt_state_cpu)

@functools.partial(jax.pmap, axis_name="data")
def train_step(params, opt_state, batch):
    def loss_fn(p):
        logits = model.apply({"params": p}, batch["image"])
        loss = optax.softmax_cross_entropy_with_integer_labels(logits, batch["label"]).mean()
        return loss

    loss, grads = jax.value_and_grad(loss_fn)(params)
    loss = jax.lax.pmean(loss, axis_name="data")
    grads = jax.lax.pmean(grads, axis_name="data")
    updates, opt_state = tx.update(grads, opt_state, params)
    params = optax.apply_updates(params, updates)
    return params, opt_state, loss

for epoch in range(args.num_epochs):
    for _ in range(args.steps_per_epoch):
        batch = next(train_iter)
        batch = shard(batch)
        params_repl, opt_state_repl, loss = train_step(params_repl, opt_state_repl, batch)
        loss = float(jax.device_get(loss)[0])
```

Model Training

JAX

```
tx = optax.adamw(args.learning_rate)
opt_state_cpu = tx.init(params_cpu)
params_repl = flax.jax_utils.replicate(params_cpu)
opt_state_repl = flax.jax_utils.replicate(opt_state_cpu)

@functools.partial(jax.pmap, axis_name="data")
def train_step(params, opt_state, batch):
    def loss_fn(p):
        logits = model.apply({"params": p}, batch["image"])
        loss = optax.softmax_cross_entropy_with_integer_labels(logits, batch["label"]).mean()
        return loss

    loss, grads = jax.value_and_grad(loss_fn)(params)
    loss = jax.lax.pmean(loss, axis_name="data")
    grads = jax.lax.pmean(grads, axis_name="data")
    updates, opt_state = tx.update(grads, opt_state, params)
    params = optax.apply_updates(params, updates)
    return params, opt_state, loss

for epoch in range(args.num_epochs):
    for _ in range(args.steps_per_epoch):
        batch = next(train_iter)
        batch = shard(batch)
        params_repl, opt_state_repl, loss = train_step(params_repl, opt_state_repl, batch)
        loss = float(jax.device_get(loss)[0])
```

explicitly replicates parameters and optimizer states across devices

Model Training

JAX

```python
tx = optax.adamw(args.learning_rate)
opt_state_cpu = tx.init(params_cpu)
params_repl = flax.jax_utils.replicate(params_cpu)
opt_state_repl = flax.jax_utils.replicate(opt_state_cpu)

@functools.partial(jax.pmap, axis_name="data")
def train_step(params, opt_state, batch):
    def loss_fn(p):
        logits = model.apply({"params": p}, batch["image"])
        loss = optax.softmax_cross_entropy_with_integer_labels(
        return loss

    loss, grads = jax.value_and_grad(loss_fn)(params)
    loss = jax.lax.pmean(loss, axis_name="data")
    grads = jax.lax.pmean(grads, axis_name="data")
    updates, opt_state = tx.update(grads, opt_state, params)
    params = optax.apply_updates(params, updates)
    return params, opt_state, loss

for epoch in range(args.num_epochs):
    for _ in range(args.steps_per_epoch):
        batch = next(train_iter)
        batch = shard(batch)
        params_repl, opt_state_repl, loss = train_step(params_repl, opt_state_repl, batch)
        loss = float(jax.device_get(loss)[0])
```

compiles the function on the first call
reuses on all devices in parallel

Model Training

JAX

```python
tx = optax.adamw(args.learning_rate)
opt_state_cpu = tx.init(params_cpu)
params_repl = flax.jax_utils.replicate(params_cpu)
opt_state_repl = flax.jax_utils.replicate(opt_state_cpu)

@functools.partial(jax.pmap, axis_name="data")
def train_step(params, opt_state, batch):
    def loss_fn(p):
        logits = model.apply({"params": p}, batch["image"])
        loss = optax.softmax_cross_entropy_with_integer_labels(logits, batch["label"]).mean()
        return loss

    loss, grads = jax.value_and_grad(loss_fn)(params)
    loss = jax.lax.pmean(loss, axis_name="data")
    grads = jax.lax.pmean(grads, axis_name="data")
    updates, opt_state = tx.update(grads, opt_state, params)
    params = optax.apply_updates(params, updates)
    return params, opt_state, loss

for epoch in range(args.num_epochs):
    for _ in range(args.steps_per_epoch):
        batch = next(train_iter)
        batch = shard(batch)
        params_repl, opt_state_repl, loss = train_step(params_repl, opt_state_repl, batch)
        loss = float(jax.device_get(loss)[0])
```

gradient syncing is explicit rather than automatic as in PyTorch DDP

Model Training

JAX

```
tx = optax.adamw(args.learning_rate)
opt_state_cpu = tx.init(params_cpu)
params_repl = flax.jax_utils.replicate(params_cpu)
opt_state_repl = flax.jax_utils.replicate(opt_state_cpu)

@functools.partial(jax.pmap, axis_name="data")
def train_step(params, opt_state, batch):
    def loss_fn(p):
        logits = model.apply({"params": p}, batch["image"])
        loss = optax.softmax_cross_entropy_with_integer_labels(logits, batch["label"]).mean()
        return loss

    loss, grads = jax.value_and_grad(loss_fn)(params)
    loss = jax.lax.pmean(loss, axis_name="data")
    grads = jax.lax.pmean(grads, axis_name="data")
    updates, opt_state = tx.update(grads, opt_state, params)
    params = optax.apply_updates(params, updates)
    return params, opt_state, loss

for epoch in range(args.num_epochs):
    for _ in range(args.steps_per_epoch):
        batch = next(train_iter)
        batch = shard(batch)
        params_repl, opt_state_repl, loss = trai
        loss = float(jax.device_get(loss)[0])
```

use shard to split data to different devices, instead of using sampler

Model Training

JAX

```
tx = optax.adamw(args.learning_rate)
opt_state_cpu = tx.init(params_cpu)
params_repl = flax.jax_utils.replicate(params_cpu)
opt_state_repl = flax.jax_utils.replicate(opt_state_cpu)

@functools.partial(jax.pmap, axis_name="data")
def train_step(params, opt_state, batch):
    def loss_fn(p):
        logits = model.apply({"params": p}, batch["image"])
        loss = optax.softmax_cross_entropy_with_integer_labels(logits, batch["label"]).mean()
        return loss

    loss, grads = jax.value_and_grad(loss_fn)(params)
    loss = jax.lax.pmean(loss, axis_name="data")
    grads = jax.lax.pmean(grads, axis_name="data")
    updates, opt_state = tx.update(grads, opt_state, params)
    params = optax.apply_updates(params, updates)
    return params, opt_state, loss

for epoch in range(args.num_epochs):
    for _ in range(args.steps_per_epoch):
        batch = next(train_iter)
        batch = shard(batch)
        params_repl, opt_state_repl, loss = train_step(params_repl, opt_state_repl, batch)
        loss = float(jax.device_get(loss)[0])
```

variables in JAX are immutable; thus, they are overwritten instead of updated

Should I Switch to JAX?

Pro

- Potentially faster; JAX's JIT is more performant than torch.compile

Con

- Lack of libraries / community support

Suggestion: use JAX if any of the following cases apply

- Have access to more TPU resources than GPU resources
- Heavy array computations are needed outside of training (e.g., RL envs)
- Prefer stateless computation (e.g., meta-learning that needs access to grad)

Community Support (based on personal experience)

PyTorch

```
torch_text_encoder = CLIPTextModel.from_pretrained(
    "stabilityai/sd-vae-ft-ema", torch_dtype=torch.bfloat16
)
```

JAX

```
jax_text_encoder = FlaxCLIPTextModel.from_pretrained(
   "stabilityai/sd-vae-ft-ema", from_pt=True,
dtype=jnp.bfloat16
)
```

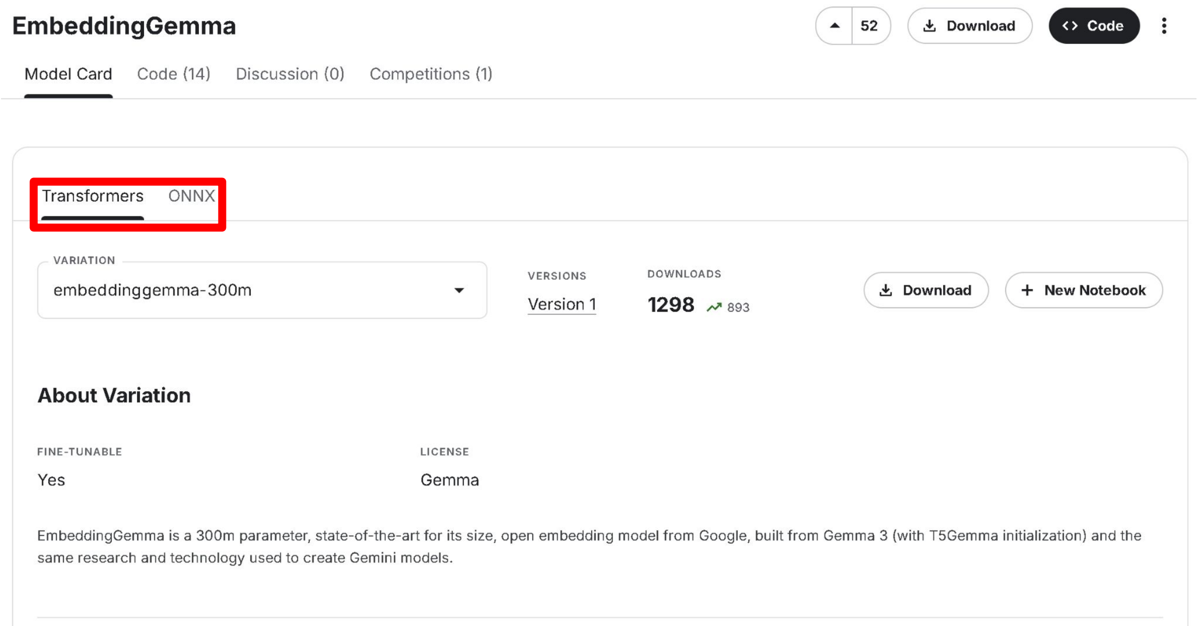
Seems easy enough…?

Community Support (based on personal experience)

```
diffusers/models
├──autoencoders/
|  ├──autoencoder_kl.py
|  ├──autoencoder_kl_qwenimage.py
|  ├──......
├──autoencoder_kl_flax.py
```

There’s often FLAX support for popular enough models / architectures/ pipelines
But recent stuff almost always need to be implemented from scratch

Community Support (based on personal experience)



Community support has generally been lacking… even for Google models

Using JAX on GPU

TPU

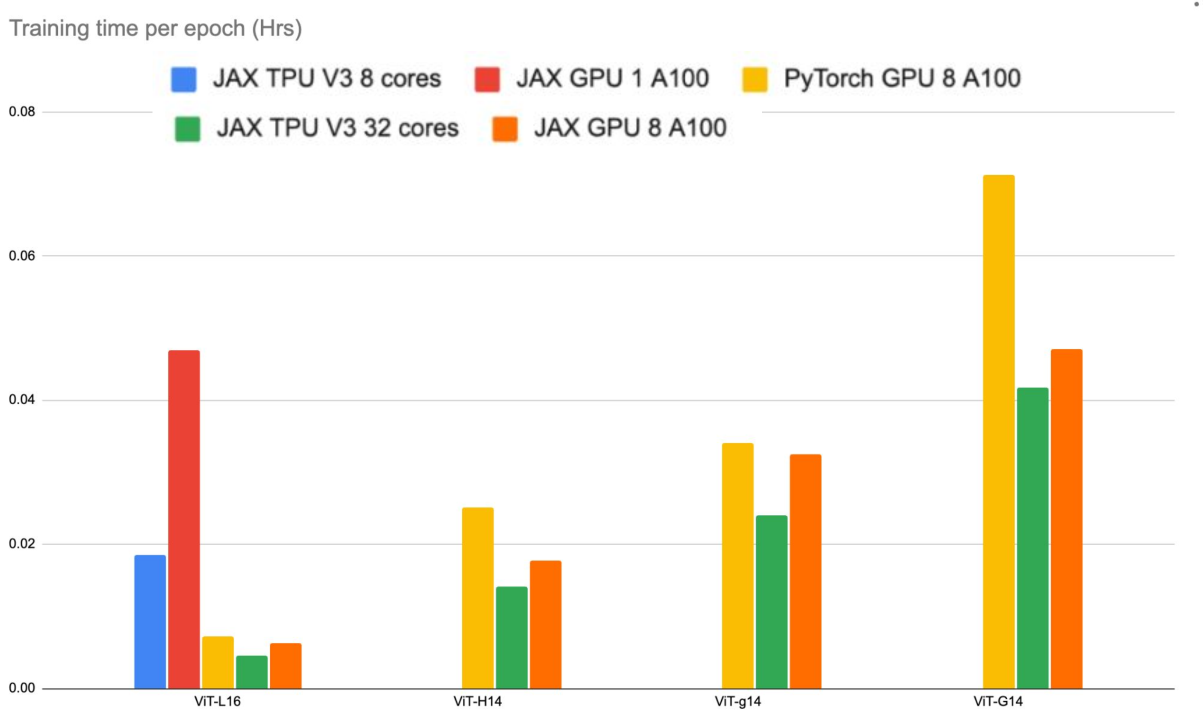
```
pip install -U "jax[tpu]"
export JAX_PLATFORMS="tpu"
```

GPU

```
pip install -U "jax[cuda12]"
export JAX_PLATFORMS="gpu"
```

Unless there's, e.g., CUDA-specific extensions, this is the only change needed

Using JAX on GPU



JAX is faster than PyTorch on GPUs for ViT training

Using PyTorch on TPU

```diff
 import torch.distributed as dist
-import torch.multiprocessing as mp
+import torch_xla
+import torch_xla.distributed.xla_backend

 def _mp_fn(rank):
   ...

-  os.environ['MASTER_ADDR'] = 'localhost'
-  os.environ['MASTER_PORT'] = '12355'
-  dist.init_process_group("gloo", rank=rank, world_size=world_size)
+  # Rank and world size are inferred from the XLA device runtime
+  dist.init_process_group("xla", init_method='xla://')
+
+  model.to('xla')
+  ddp_model = DDP(model, gradient_as_bucket_view=True)

-  model = model.to(rank)
-  ddp_model = DDP(model, device_ids=[rank])

   for inputs, labels in train_loader:
+    with torch_xla.step():
+      inputs, labels = inputs.to('xla'), labels.to('xla')
       optimizer.zero_grad()
       outputs = ddp_model(inputs)
       loss = loss_fn(outputs, labels)
       loss.backward()
       optimizer.step()

 if __name__ == '__main__':
-  mp.spawn(_mp_fn, args=(), nprocs=world_size)
+  torch_xla.launch(_mp_fn, args=())
```

running PyTorch on TPU requires using the torch_xla library

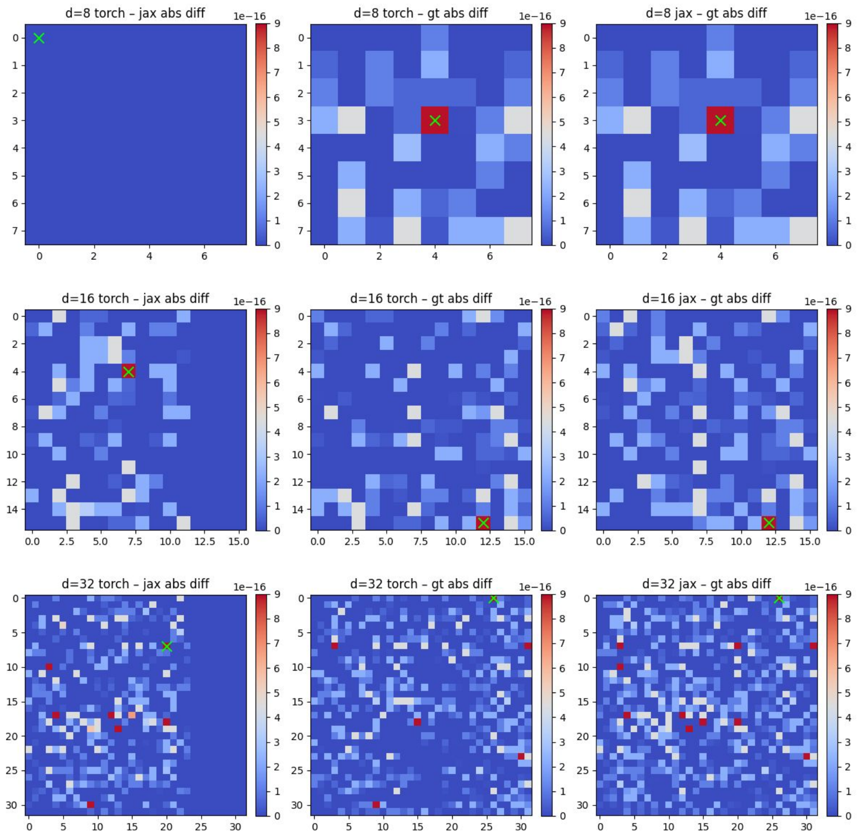
Numerical Inconsistencies b/w JAX & PyTorch

Differences from JAX

Our models were originally trained in JAX on TPUs. The weights in this repo are ported directly from the JAX models. There may be minor differences in results stemming from sampling with different floating point precisions. We re-evaluated our ported PyTorch weights at FP32, and they actually perform marginally better than sampling in JAX (2.21 FID versus 2.27 in the paper).

a note from the DiT codebase

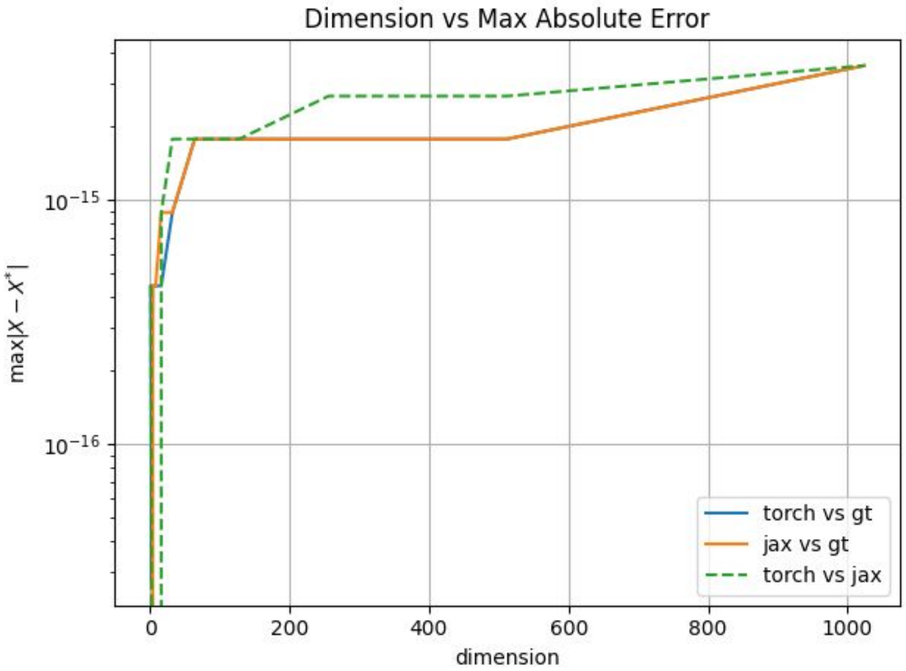
Numerical Inconsistencies b/w JAX & PyTorch



[d,4] x [4,d] matmul in JAX vs. PyTorch

Inconsistency exists and increase is larger in larger matrices

Numerical Inconsistencies b/w JAX & PyTorch

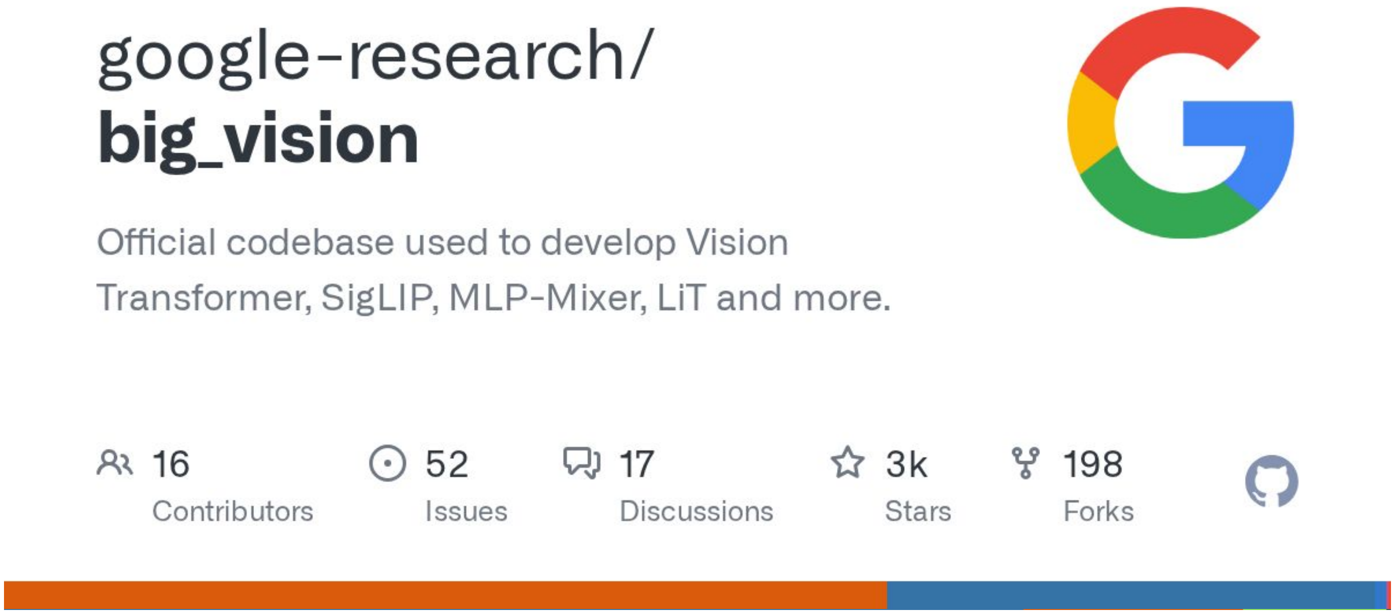


[d,4] x [4,d] matmul in JAX vs. PyTorch

Inconsistency exists and increase is larger in larger matrices

GT is from a high-precision library

Repos - Big Vision



Repos - Big Vision

AI-Hypercomputer/
maxtext

A simple, performant and scalable Jax LLM!

167 Contributors

65 Issues

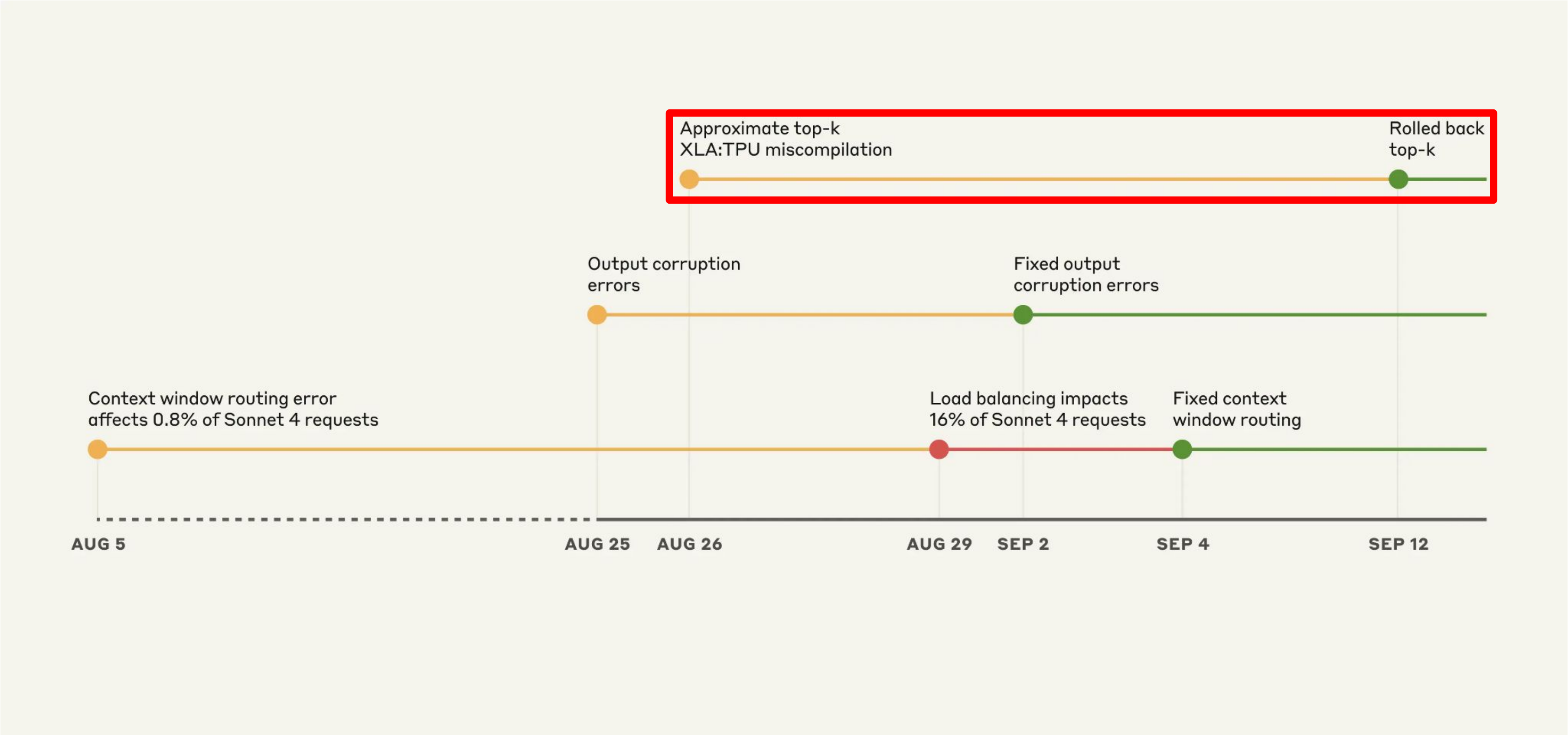
2k Stars

417 Forks

Issues of JAX/TPU: the Intriguing Case of Anthropic

“We deploy Claude across multiple hardware platforms, namely AWS Trainium, NVIDIA GPUs, and Google TPUs. This approach provides the capacity and geographic distribution necessary to serve users worldwide.”

Issues of JAX/TPU: the Intriguing Case of Anthropic



Issue 1: TPU implementation would occasionally drop the most probable token when greedy sampling

Explanation: the vector processor is fp32-native, so the TPU compiler (XLA) can optimize runtime by converting some operations to fp32

So different parts of the system “disagreed” about which token was highest, since they were running at different precision levels

Solution: disable using higher precision when compiling operations (“xla_allow_excess_precision”)

Issue 2: approximate top-k operation occasionally return completely wrong results

jax.lax.approx_max_k

```
jax.lax.approx_max_k(operand, k, reduction_dimension=-1, recall_target=0.95,
reduction_input_size_override=-1, aggregate_to_topk=True)
```

Returns max `k` values and their indices of the `operand` in an approximate manner.

"The bug's behavior was frustratingly inconsistent. It changed depending on unrelated factors such as what operations ran before or after it, and whether debugging tools were enabled. The same prompt might work perfectly on one request and fail on the next."

Solution: switch to exact top-k operation

Thank you!

Questions?

Visit github.com/TaiMingLu/TPU-Manual for our TPU Manual

Acknowledgement:
Thanks @Yufeng Xu for contributing to the slides