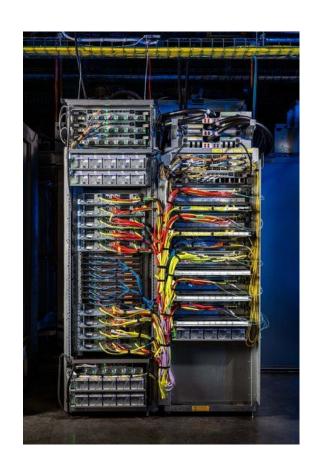
# Building Distributed ML Supercomputers

- 1. Introduction and Agenda
- 2. GPU Refresher and TPU 101
- 3. Frameworks and Orchestration
- 4. Learning Material
- 5. QnA

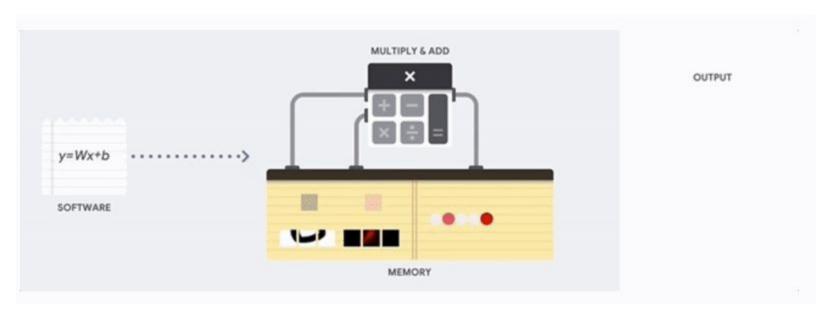


Erik Saarenvirta (esaarenvirta@)
Staff Technical Solution Consultant



## **GPU Refresher and TPU 101**

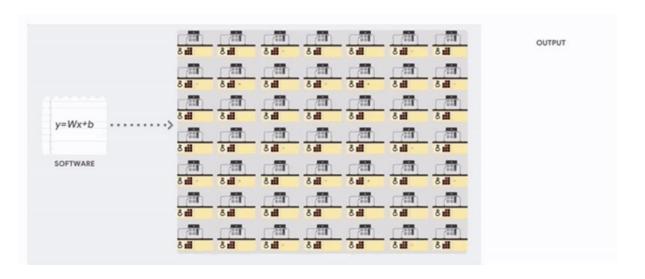
#### Refresher: How does a CPU work?



**CPU:** A CPU loads values from memory, performs a calculation on the values and stores the result back in memory for every calculation. Memory access is slow when compared to the calculation speed and can limit the total throughput of CPUs. This is often referred to as the <u>von Neumann bottleneck</u>.



#### Refresher: How does a GPU Work?



**GPU:** Tackles the memory access bottleneck with brute force and **massive parallelism**. It *still runs back and forth to memory*, but it does it with thousands of

ALUs at the same time, making it great

Arithmetic Logic Units (ALUs) in a

**GPUs** contain thousands of

single processor.

for tasks that can be broken up (Neural networks)

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



#### What is a TPU?

Tensor Processing Unit





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

Custom accelerator chip by Google to train and execute deep neural networks

Whitepaper: In-Datacenter
Performance Analysis of a Tensor
Processing Unit

Article: TPU v2

Blog post outlining most of this

presentation: <u>Here</u>



## Why did we design the TPU (v1)?

Designed for inference at the start

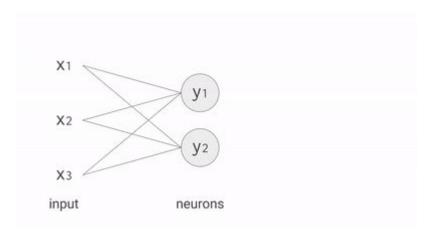
**Inference:** The process of running a trained neural network to classify data with labels or estimate some missing or future values

#### **Inference Calculations:**

**Multiply** input data (x) with model weights (w) to represent the signal strength

**Add** the results to aggregate the neuron's state into a single value

Apply an **activation** function (f) (such as <u>ReLU</u>, <u>Sigmoid</u>, <u>tanh</u> or others) to modulate the artificial neuron's activity.

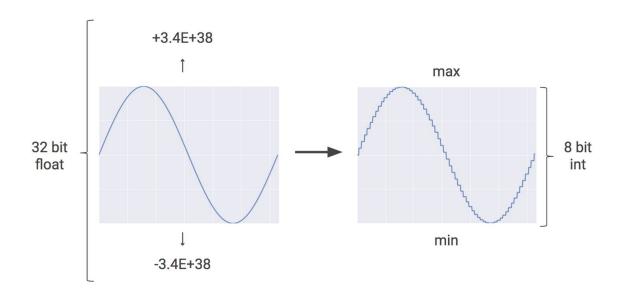


A neural network takes input data, multiplies them with a weight matrix and applies an activation function



#### Quantization

TPU v1 was only int8



A TPU v1 contains 65,536 8-bit integer multipliers

At the time, a comparable GPU had a few thousand 32-bit floating point multipliers

As long as your use case could maintain accuracy in int8, that's a large improvement in performance



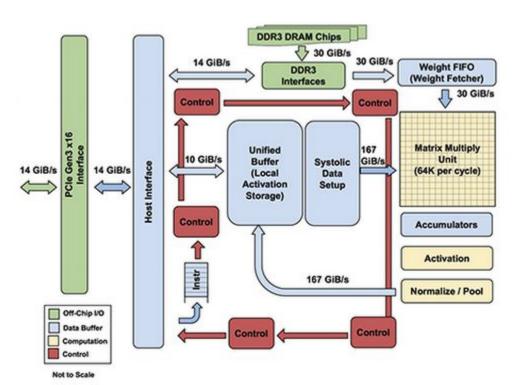
#### **TPU Instruction Set**

Most modern CPUs are heavily influenced by the <u>Reduced</u> <u>Instruction Set Computer (RISC)</u> design → simple instructions (e.g., load, store, add and multiply)

TPU chose the <u>Complex Instruction Set Computer (CISC)</u> style as the basis of the TPU instruction set instead → high-level instructions that run more complex tasks (e.g multiply-and-add many times)



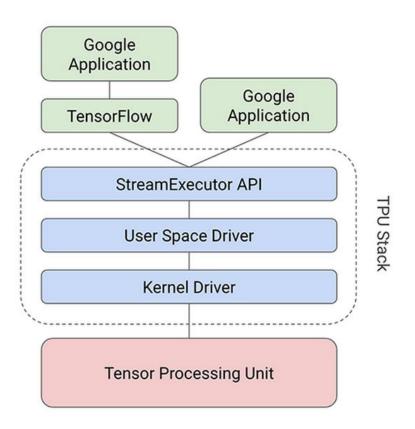
#### **TPU Instruction Set**



TPU Instruction	Function
Read_Host_Me mory	Read data from memory
Read_Weights	Read weights from memory
MatrixMultiply/ Convolve	Multiply or convolve with the data and weights, accumulate the results
Activate	Apply activation functions
Write_Host_Me mory	Write result to memory



## **TPU programmability**



TPU design allows for programming a wide variety of neural network models

We created a compiler and software stack that translates API calls from TensorFlow graphs into TPU instructions

**MXU** architecture

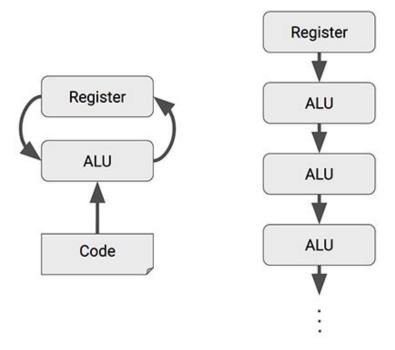
RISC  $\rightarrow$  Simple instructions for multiplying or adding  $\rightarrow$  Scalar processors (e.g single operation with each instruction)

CPUs have clock cycles in the gigahertz range but still take a while to compute a matrix via scalar operations e.g 100s to 1000s operations per clock cycle via vector operation instruction set extensions

We designed an MXU (Matrix Multiplier Unit) that could process hundreds of thousands of operations in a clock cycle. Features a different architecture from CPUs and GPUs called a **Systolic Array** 



#### **MXU** architecture



(left side) CPU/GPUs are general purpose → store values in registers, program tells ALUs which registers to read, which op to perform and which register to put the result → requires energy to access multiple registers per op

(right side) A systolic array chains multiple ALUs together, reusing the result of reading a single register.



#### **MXU** architecture



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

#### MXU:

Can read each input once, but use it for many different operations without storing it back to a register.

Wires only connect spatially adjacent ALUs, which makes them short and energy-efficient.

The ALUs perform only multiplications and additions in fixed patterns, which simplifies their design.



#### **MXU** architecture



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

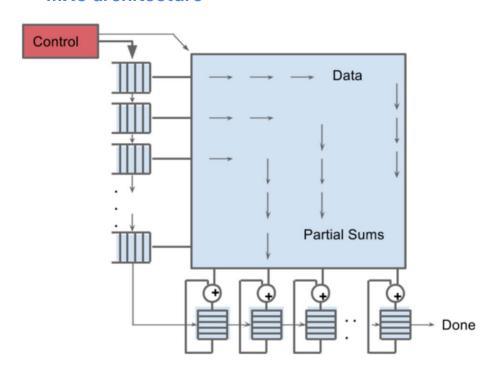
Systolic → data flows in waves, like a heart pumping blood

Systolic array in the MXU → optimized for power and area efficiency in performing matrix multiplications

Engineering trade off → Limited registers, control and operational flexibility in exchange for efficiency and operation density while doing *matrix multiplication* and not being well suited for general computation



#### **MXU** architecture



TPU v1 MXU contains  $256 \times 256 = \text{total}$  65,536 ALUs.

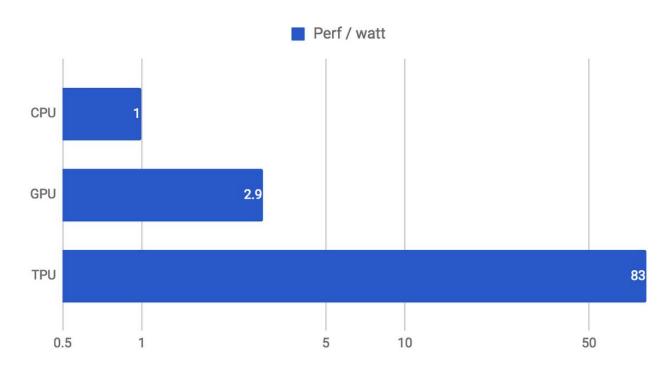
Can process 65,536 multiply-and-adds for 8-bit integers every cycle.

A TPU runs at 700MHz  $\rightarrow$  Can compute 65,536  $\times$  700,000,000 = 46  $\times$  1012 multiply-and-add operations or 92 Teraops per second (92  $\times$  1012) in the matrix unit.

A single MatrixMultiply cycle → 100s of 1000s of ops → Intermediate results passed between all 65K ALUs without memory access

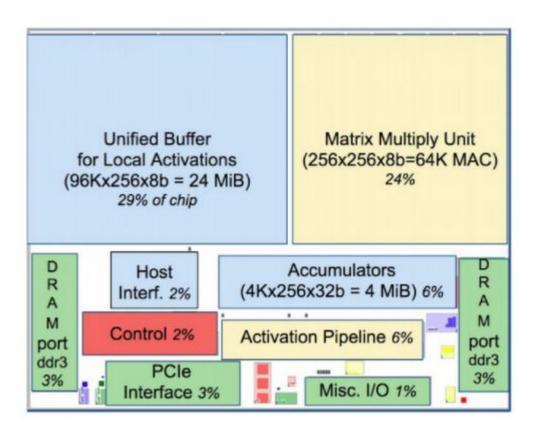


## **TPU Performance per Watt comparison (2017)**





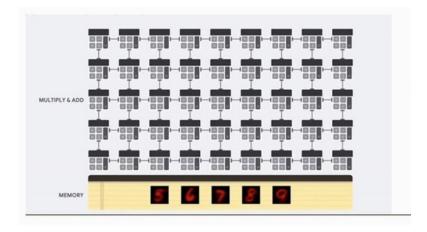
## Floor plan of a TPU V1

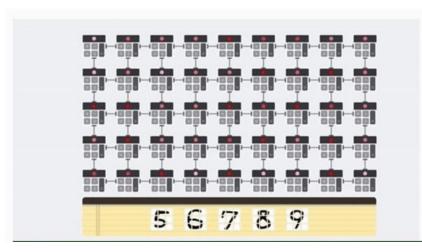


floor plans of CPUs and GPUs → red parts (control logic) are much larger (and thus more difficult to design) for CPUs and GPUs since they need to realize the more complex constructs and mechanisms

In the TPU, the control logic is minimal and takes under 2% of the die.

## Comparing to before: TPU summary





OUTPUT

**TPU:** Designed to *eliminate* the memory access bottleneck for its **one specific job** (matrix multiplication). Data and parameters are loaded into the **Matrix Multiplication Unit (MXU)**, and then the results flow *directly* between the accumulators without needing to access memory again until the final result is ready

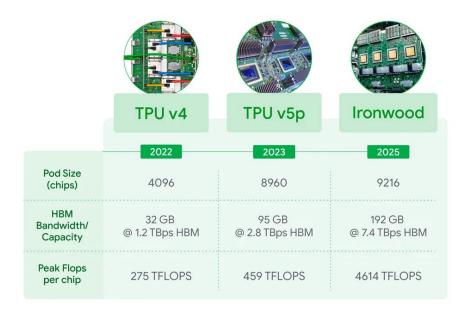
- TPU loads model parameters from High Bandwidth Memory (HBM) into the MXU.
- TPU loads data from HBM. As each multiplication is executed, the result is passed to the next multiply-accumulator. The output is the summation of all multiplication results between the data and parameters.

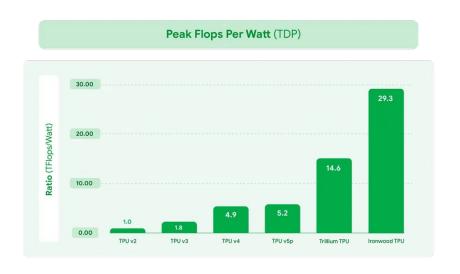
This is what makes them great for neural networks

#### **TPU evolution (Back to 2025)**

2015 2018 2020 2022 2023 2024 2025 v2 Trillium v1**v**3 Ironwood v4 v5e v5p 1x/chip 1x/chip 3x/chip 6.6x/chip 4x/chip 21x/chip 100x v2 TPU7x: 9,216 chips/pod inference 1x/pod 12x/pod 100x/pod inference 750x/pod performance TPU7: 256 chips/pod Internal inference Distributed Liquid cooled Optically Cost-efficiency Most flexible Enabling the Cutting-edge chip next frontier of accelerator shared memory reconfigurable for large-scale Al accelerator training and Al models Largest pod inference

#### **TPU** evolution

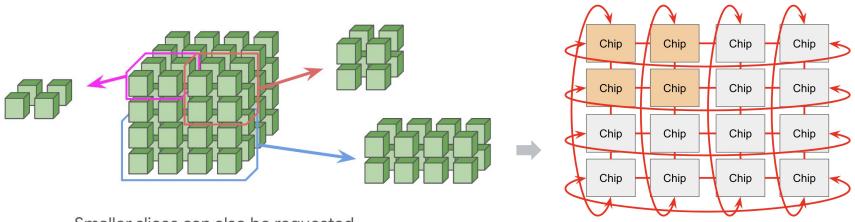






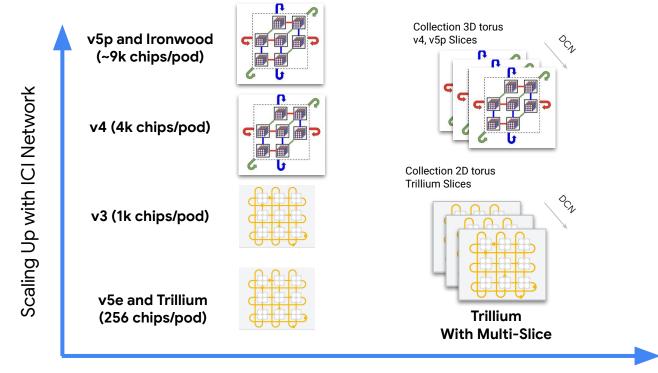
#### **TPU Pod & Slice**

A TPU Pod is a single, unified package of multiple TPU chips interconnected via the Inter-Chip Interconnect (ICI). The number of TPU chips contained in a TPU Pod varies depending on the TPU version (TPU v6e: 256, TPU v7: 256/9,216).



Smaller slices can also be requested.

## TPUs allow Google to massively scale up & out

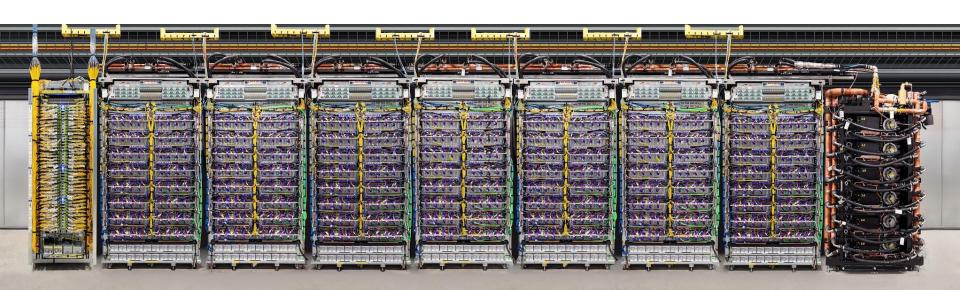


Scaling Out with Data Center Network

9,216 chips

42.5

Exaflops per pod



# Frameworks and Orchestration

#### What is the JAX AI Stack?

A curated set of interoperable libraries for high-performance ML research and development.

- Core Philosophy: Achieve Performance, Flexibility, and Scalability using function transformations.
- Engine: Uses the XLA (Accelerated Linear Algebra)
   compiler to generate highly optimized code for the target
   hardware.
- Portability: Enables running the same code, often without modification, across CPUs, GPUs, and TPUs.



## Why did Google develop JAX?

- Google learned from years of experience
  - DistBelief
  - TensorFlow
- Google needed high performance to scale efficiently
- Google needed flexibility and modularity to innovate quickly

High performance, flexibility, and modularity became the guiding principles for the development of JAX

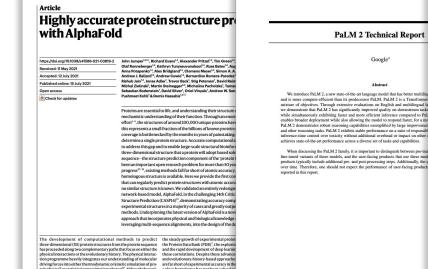


### Flexibility and Modularity

- You've seen results generated with JAX
- Google uses JAX for nearly all of its research and GenAI development
- Gemini, Gemma, Imagen, Veo, Waymo, etc. are all created using JAX

PaLM 2 Technical Report

Google\*



#### Google An overview of Bard: an early experiment with g James Manylka, SVP, Research, Technology and Society, and Sissle Hslao, Vice President and General M Experiment Updates log or read more on the Google Keyword blog. We have long seen the notential of Al to make information and computing more accepeople. As part of this journey, we have made pioneering advancements on large land and have seen great progress across Google and in this field more broadly. For several applied LLMs in the background to improve many of our products, such as autocomp Gmail, expanding Google Translate, and helping us better understand queries in Goog using LLMs to power Bard, an experiment that allows people to collaborate directly w While we're at an important inflection point and encouraged by the widespread excite

generative Al, it's still early days for this technology. The following outlines how we are on Bard - what it is, how it works and its current capabilities and limitations. Our app as Bard itself (and its underlying technology) does, and as we learn from ongoing rese

#### What Bard is

Bard is designed as an interface to an LLM that enables users to collaborate with gene one of the promises of LLM-based innovations like Bard is to help people unlock their they can augment their imagination, expand their curiosity, and enhance their produc

We launched Bard as an experiment in March 2023. Since then, we have iterated quick capabilities - always in accordance with our Al Principles. We continue to engage with educators, policymakers, civil rights and human rights leaders, content creators and c the many possible applications, as well as the risks and limitations, of this emerging te

We think Bard is most helpful right now as a standalone experiment. It best allows us

Google DeepMind

#### Gemini: A Family of Highly Capable Multimodal Models

Gemini Team, Google<sup>1</sup>

This report introduces a new family of multimodal models, Gemini, that exhibit remarkable capabilities across image, audio, video, and text understanding. The Gemini family consists of Ultra, Pro, and Nano sizes, suitable for applications ranging from complex reasoning tasks to on-device memory-constrained use-cases. Evaluation on a broad range of benchmarks shows that our most-capable Gemini Ultra model advances the state of the art in 30 of 32 of these benchmarks - notably being the first model to achieve human-expert performance on the well-studied exam benchmark MMLU, and improving the state of the art in every one of the 20 multimodal benchmarks we examined. We believe that the new capabilities of Gemini models in cross-modal reasoning and language understanding will enable a wide variety of use cases and we discuss our approach toward deploying them responsibly to users.

We present Gemini, a family of highly capable multimodal models developed at Google. We trained Gemini jointly across image, audio, video, and text data for the purpose of building a model with both strong generalist capabilities across modalities alongside cutting-edge understanding and reasoning performance in each respective domain.

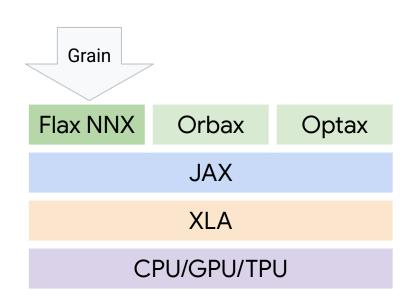
Gemini 1.0, our first version, comes in three sizes: Ultra for highly-complex tasks, Pro for enhanced performance and deployability at scale, and Nano for on-device applications. Each size is specifically tailored to address different computational limitations and application requirements. We evaluate the performance of Gemini models on a comprehensive suite of internal and external benchmarks covering a wide range of language, coding, reasoning, and multimodal tasks.

Gemini advances state-of-the-art in large-scale language modeling (Anil et al., 2023; Brown et al., 2020; Chowdhery et al., 2023; Hoffmann et al., 2022; OpenAI, 2023a; Radford et al., 2019; Rae et al., 2021), image understanding (Alayrac et al., 2022; Chen et al., 2022; Dosovitskiy et al., 2020; OpenAI, 2023b; Reed et al., 2022; Yu et al., 2022a), audio processing (Radford et al., 2023; Zhang et al., 2023), and video understanding(Alayrac et al., 2022; Chen et al., 2023). It also builds on the work on sequence models (Sutskever et al., 2014), a long history of work in deep learning based on neural networks (LeCun et al., 2015), and machine learning distributed systems (Barham et al., 2022; Bradbury et al., 2018; Dean et al., 2012) that enable large-scale training.

Our most capable model, Gemini Ultra, achieves new state-of-the-art results in 30 of 32 benchmarks

## The Full Stack: A Modular, Layered Design

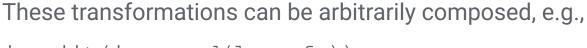
- Grain: Data Loading (optional)
- Flax NNX: Neural Networks
- Optax: Optimizers
- Orbax: Checkpointing
- JAX: Function Transformations & NumPy API
- XLA: Compiler



## The JAX Engine: Composable Function Transformations

JAX's power comes from wrapping Python functions in transformations that change *how* they execute.

- jax.jit() -> Compiles the function with XLA for high speed.
- jax.grad() -> Creates a new function that computes gradients.
- jax.vmap() -> **Vectorizes** or "auto-batches" the function.



```
jax.jit(jax.grad(loss_fn)).
```



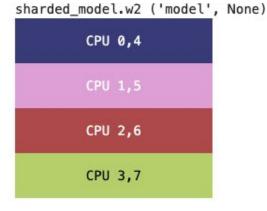
### JAX Superpower: Flexible Parallelism

JAX offers powerful, flexible ways to scale across multiple accelerators, driven by the compiler.

#### PyTorch: Library-Based

- You wrap your single-device model in a library object like DDP or FSDP.
- The library manages communication behind the scenes.
- model = DDP (model)

#### JAX: Compiler-Driven (SPMD)



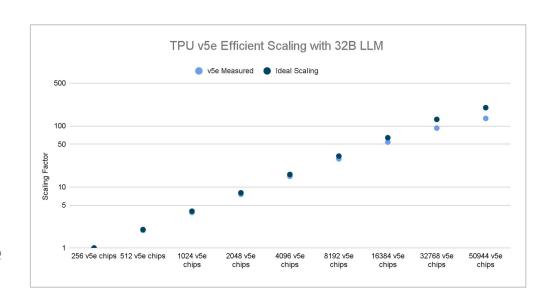
- You describe the desired parallel layout of your data and parameters using sharding annotations.
- jax.jit compiles a new, optimized parallel program from scratch based on these annotations.
- This provides a more unified and flexible approach to different parallelism strategies (Data, Model, etc.).

## JAX Scalability: Scaling to 50,944 TPUs with JAX

#### JAX Scalability: TPUs

In November 2023, we used Multislice Training to run an extremely large LLM distributed training job

- 50,944 Cloud TPU v5e chips (spanning 199 Cloud TPU v5e pods)
- Near ideal scaling

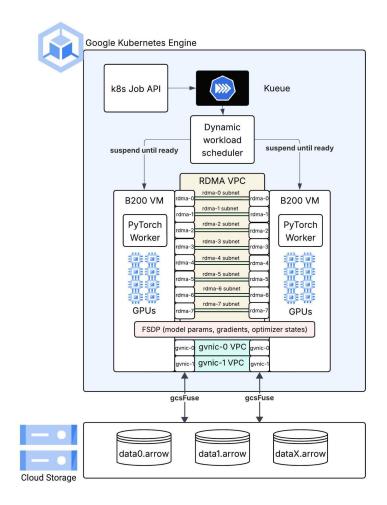


#### **Orchestration**

Frameworks (PyTorch, Jax, etc) and Cloud
Providers manage a lot of the hard work for us so
a simplified view of the problem is building
distributed infrastructure on our choice of
platform and running our launch commands in a
distributed environment. To the right is a
simplified architecture of running PyTorch on k8s.

# torchrun --nnodes=\$NUM\_NODES --nproc-per-node=\$NUM\_TRAINERS --max-restarts=3 --rdzv-id=\$JOB\_ID --rdzv-backend=c10d --rdzv-endpoint=\$HOST\_NODE\_ADDR YOUR\_TRAINING\_SCRIPT.py (--arg1 ... train script args...)

Google Cloud

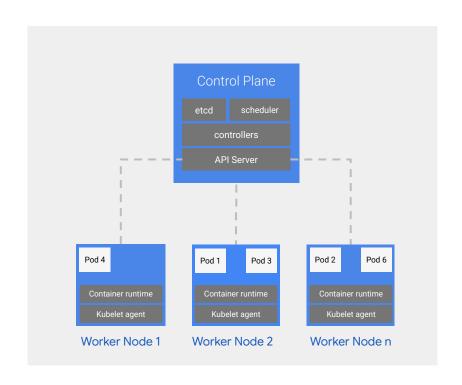


#### **Kubernetes for Orchestration**

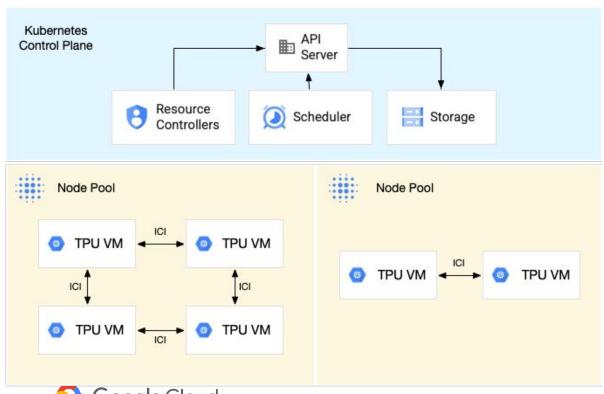
A Kubernetes cluster is composed of several pieces:

- A control plane, responsible of handling the overall status of the cluster. Includes components like:
  - etcd database
  - scheduler
  - controllers
  - API server
- Worker nodes, responsible for running user workloads and management components (container runtime, kubelet)





#### **Kubernetes for Orchestration**



(left) A TPU v5e has hosts (VMs) containing 4 chips. We can see a nodepool that has a 4x4 pod-slice deployed and one with 2x4.

We use k8s the native way, and can reference TPUs in our Jobs or other k8s supported APIs:

resources:

requests:

google.com/tpu: 4

limits:

google.com/tpu: 4



#### How do customers do this in real life?

(The bad way) Writing a lot of bash scripts using cloud provider APIs to build clusters, networks, other infrastructure

#### Here's a tutorial showing how that would work:

https://github.com/ai-on-gke/tutorials-and-examples/blob/main/dws-flex-training-pytorch/setup/setup.sh

(The good way) Using frameworks like <u>Terraform</u> to build infrastructure as code so we have a deterministic way to tear down and rebuild infrastructure

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



#### TPU has a few shortcuts

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

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



### Hello world parallelism example

```
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.

## Hello world parallelism example

```
gcloud compute tpus tpu-vm ssh my-tpu-pod-name \
--zone=us-central2-b \
--worker=all \
--command="python3 my_parallel_tutorial.py"
```

When running Jax in "Multi Controller" mode there is no "main" node. You must run the exact same python script on every single host in the cluster simultaneously. The TPU pod is configured to 'magically' handle the communication and identification of TPU devices to make each chip seamlessly work together

If you are using Google Cloud (GCP), you use the --worker=all flag to broadcast the command to all hosts managing the 256 chips.

## **Job Opportunities**

# **Learning Material**

## Google Cloud TPU Documentation

https://docs.cloud.google.com/tpu/docs/intro-to-tpu

And all content along the side bar

## Learning Resources for Jax

Code Exercises, Quick References, and Slides

https://goo.gle/learning-jax



#### MaxText

https://github.com/Al-Hypercomputer/maxtext