

Building Distributed ML Supercomputers

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

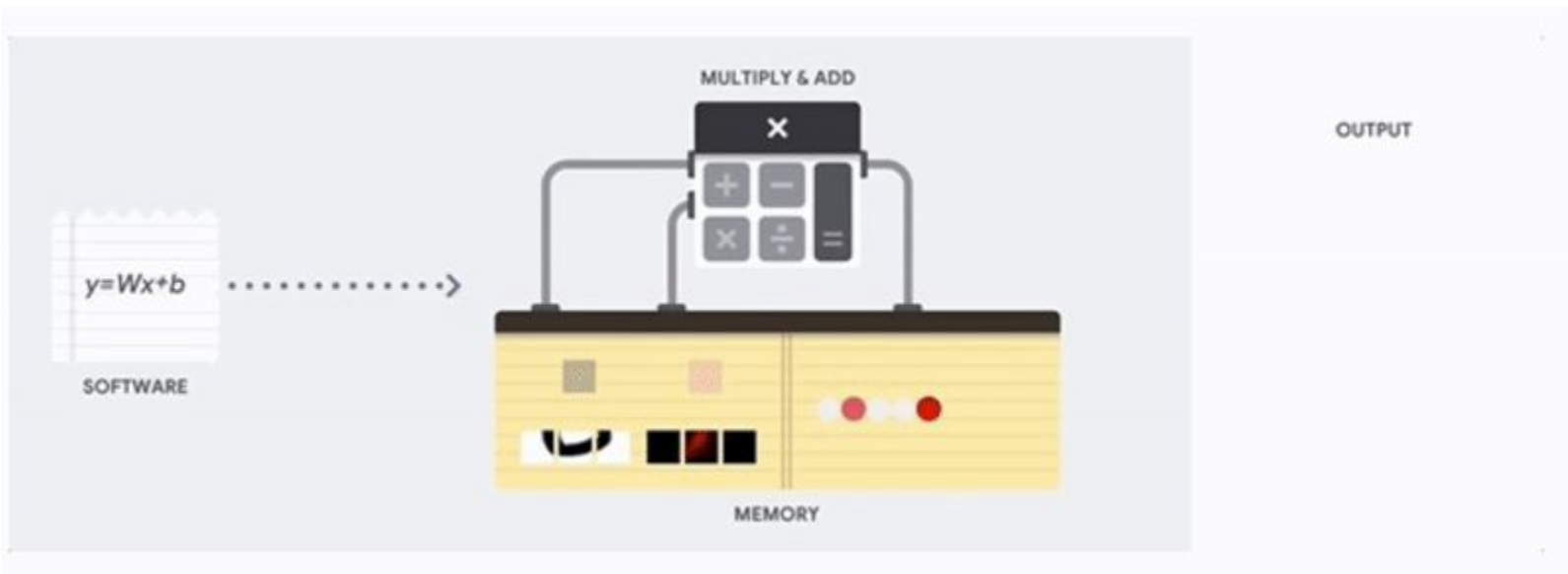


Erik Saarevirta (esaarevirta@)
Staff Technical Solution Consultant



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 von Neumann bottleneck.

Refresher: How does a GPU Work?



GPUs contain thousands of Arithmetic Logic Units (ALUs) in a single processor.

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 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: [Here](#)

Why did we design the TPU (v1)?

Designed for inference at the start

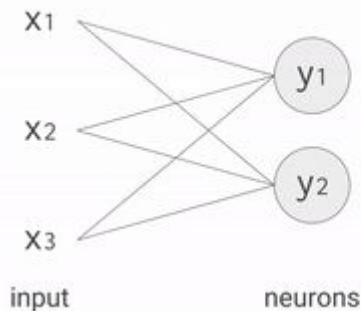
Inference: The process of running a trained neural network to classify data with labels or estimate some missing or future values

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

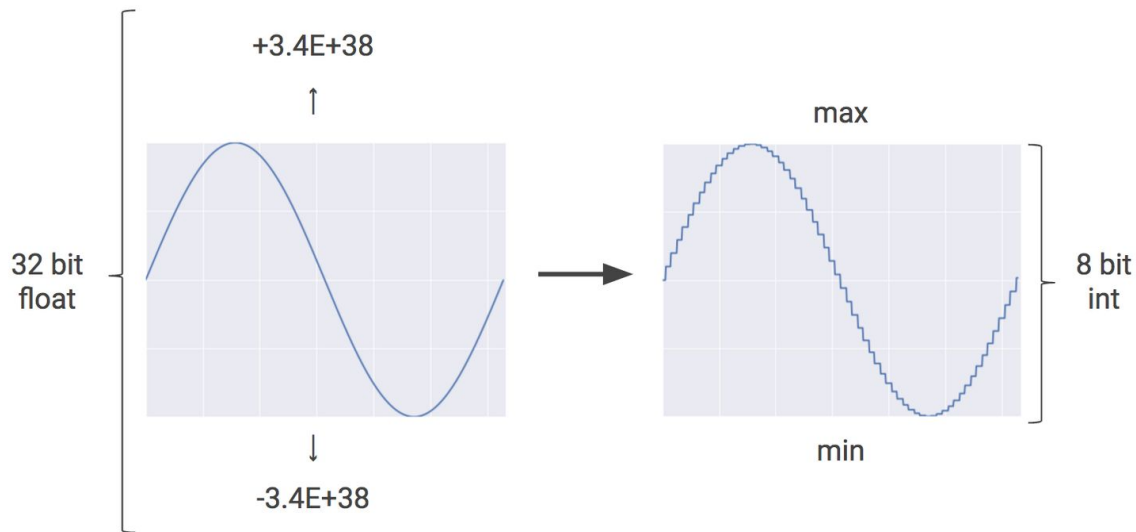
Apply an **activation** function (f) (such as [ReLU](#), [Sigmoid](#), [tanh](#) or others) to modulate the artificial neuron's activity.



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

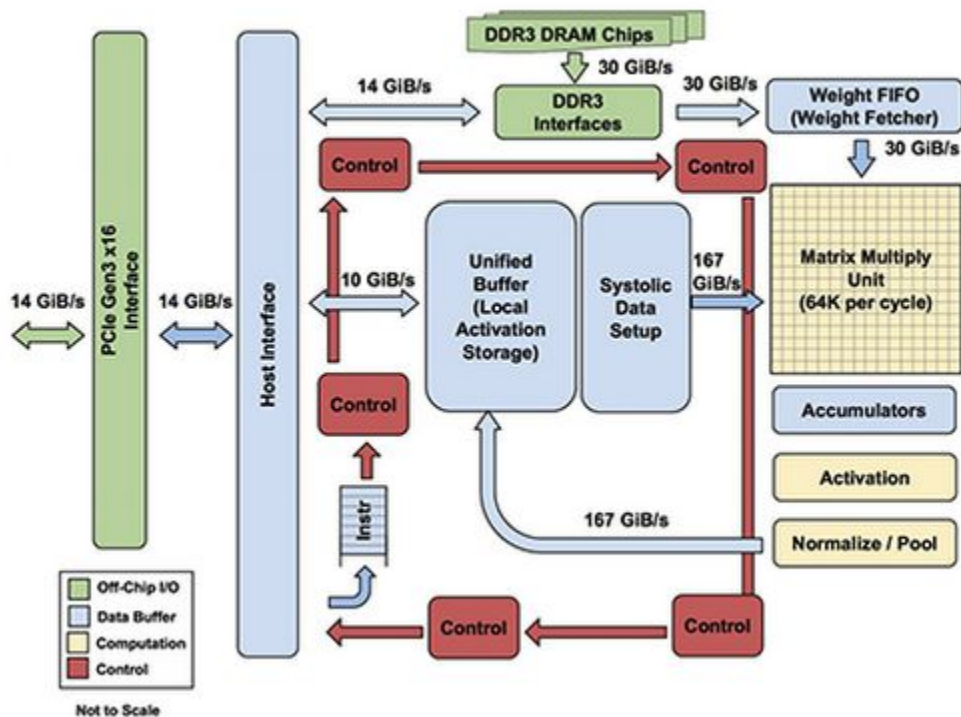
As long as your use case could maintain accuracy in int8, that's a large improvement in performance

TPU Instruction Set

Most modern CPUs are heavily influenced by the [Reduced Instruction Set Computer \(RISC\)](#) design → simple instructions (e.g., load, store, add and multiply)

TPU chose the [Complex Instruction Set Computer \(CISC\)](#) 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

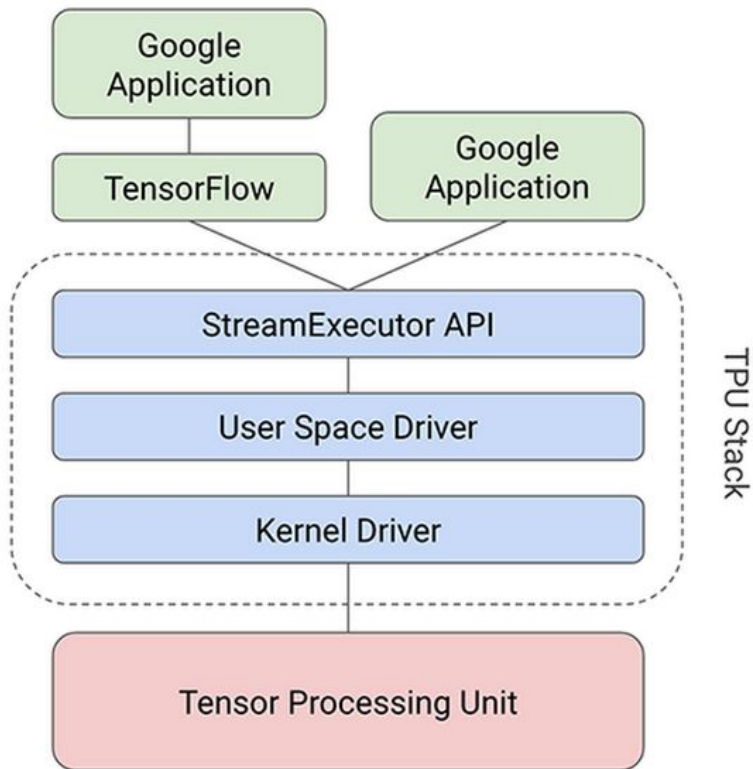


TPU Instruction

Function

Read_Host_Memory	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_Memory	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

Systolic Arrays

MXU architecture

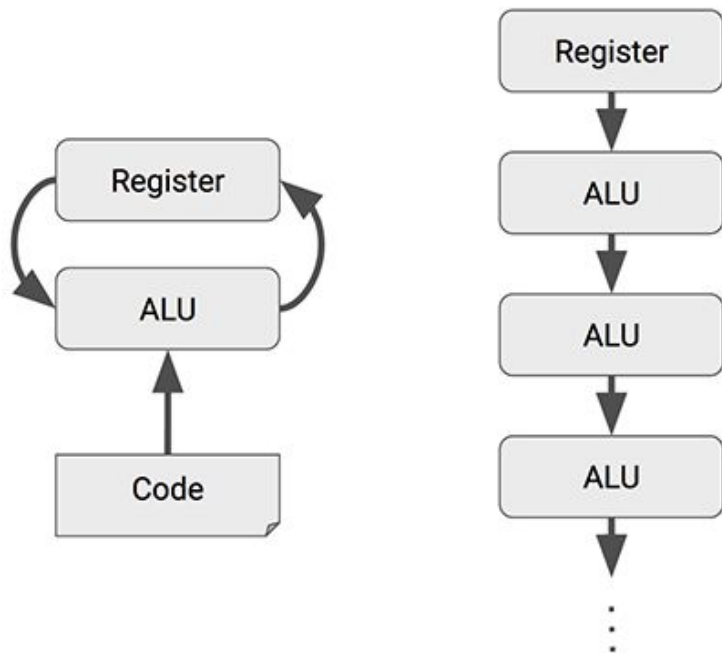
RISC → Simple instructions for multiplying or adding → Scalar processors (e.g single operation with each instruction)

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**

Systolic Arrays

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.

Systolic Arrays

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.

Systolic Arrays

MXU architecture



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

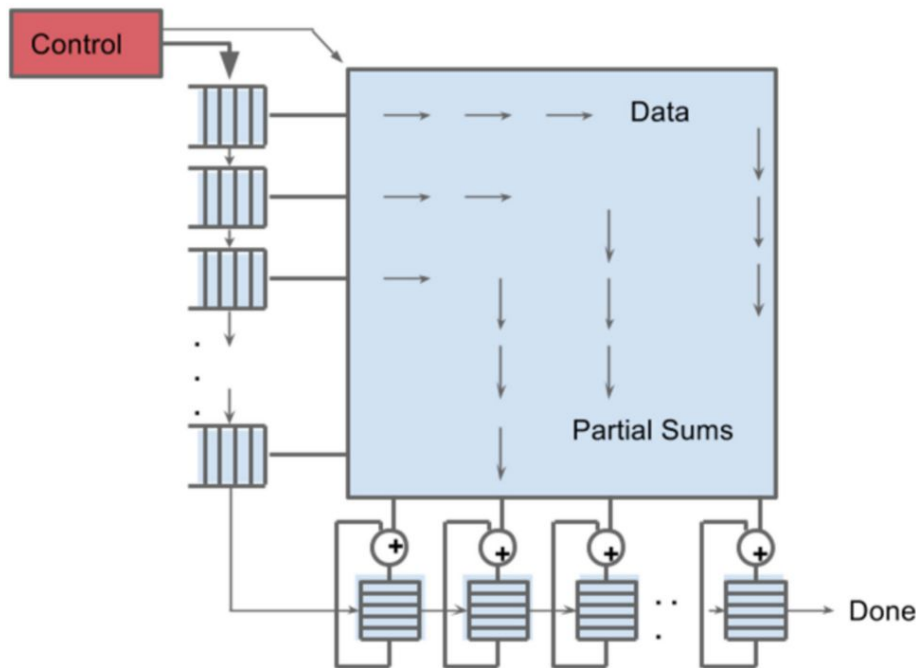
Systolic → data flows in waves, like a heart pumping blood

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

Systolic Arrays

MXU architecture



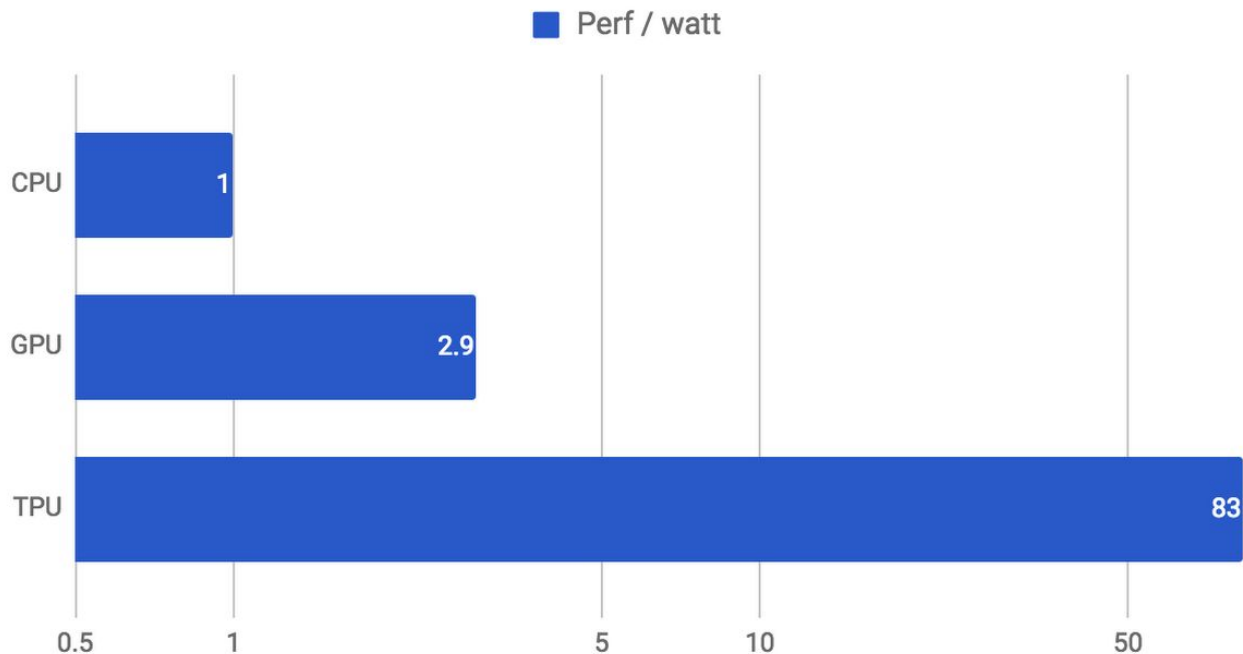
TPU v1 MXU contains $256 \times 256 =$ total 65,536 ALUs.

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

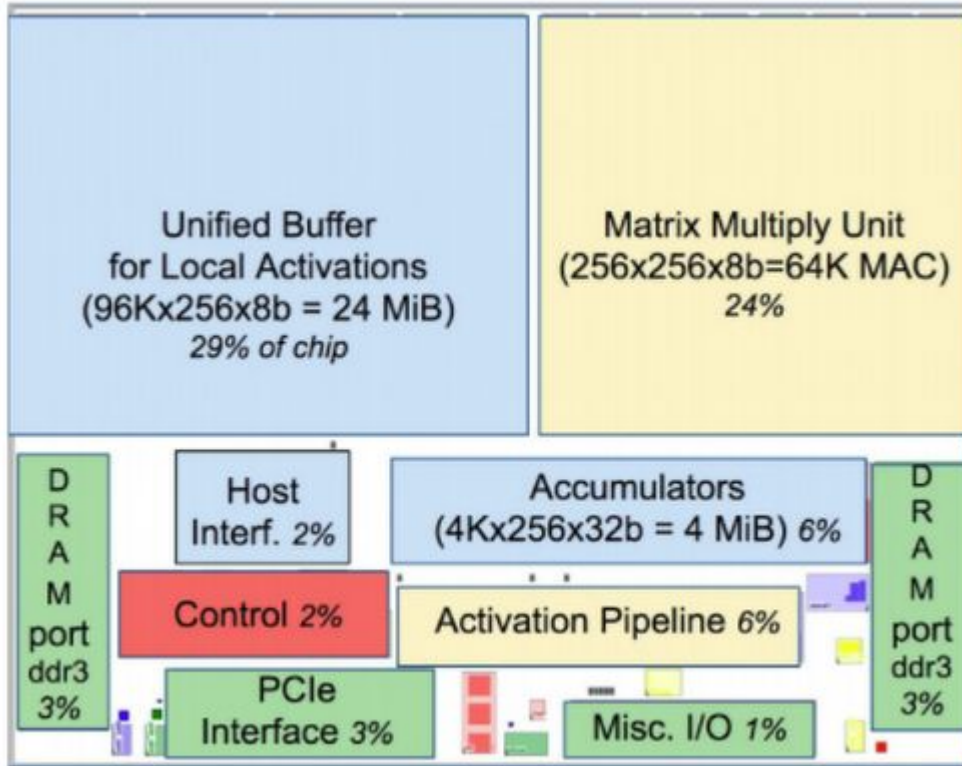
A TPU runs at 700MHz \rightarrow Can compute $65,536 \times 700,000,000 = 46 \times 10^{12}$ multiply-and-add operations or 92 Teraops per second (92×10^{12}) in the matrix unit.

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

TPU Performance per Watt comparison (2017)



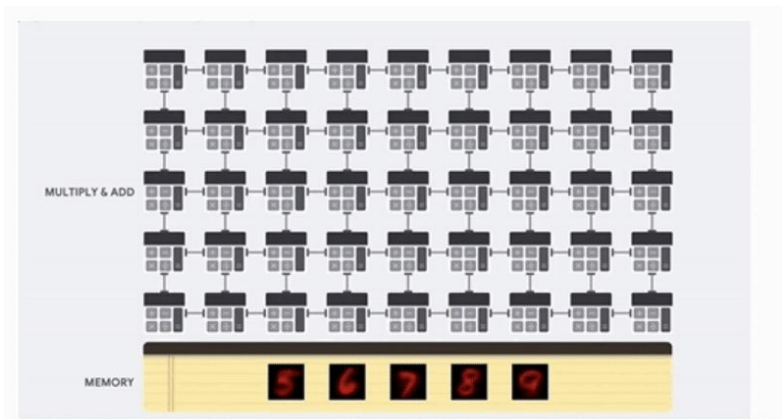
Floor plan of a TPU V1



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.

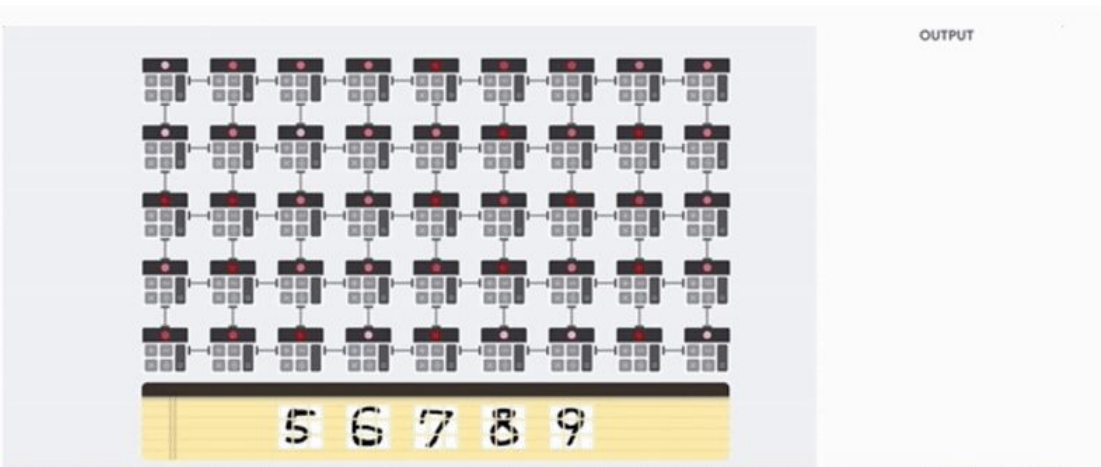
Comparing to before: TPU summary



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

- 1) TPU loads model parameters from **High Bandwidth Memory (HBM)** into the **MXU**.
- 2) 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

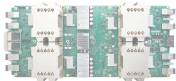


v1

1x/chip
inference

Internal inference
accelerator

2018



v2

1x/chip
1x/pod

Distributed
shared memory

2020

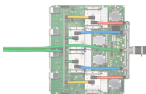


v3

3x/chip
12x/pod

Liquid cooled

2022



v4

6.6x/chip
100x/pod

Optically
reconfigurable

2023



v5e

4x/chip
inference

Cost-efficiency
for large-scale
training and
inference



v5p

21x/chip
750x/pod

Most flexible
AI accelerator

2024



Trillium

100x v2
performance

Enabling the
next frontier of
AI models

2025



Ironwood




TPU7x: 9,216 chips/pod

TPU7: 256 chips/pod

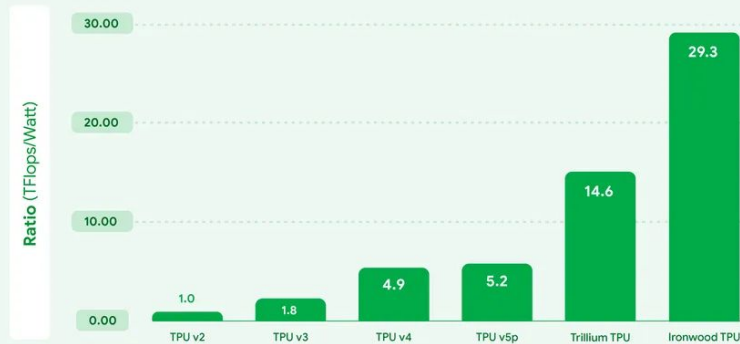
Cutting-edge chip

Largest pod

TPU evolution

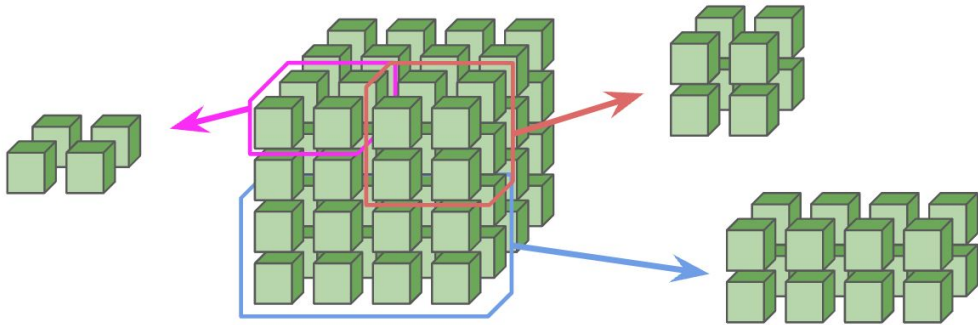
	 TPU v4	 TPU v5p	 Ironwood
	2022	2023	2025
Pod Size (chips)	4096	8960	9216
HBM Bandwidth/ Capacity	32 GB @ 1.2 TBps HBM	95 GB @ 2.8 TBps HBM	192 GB @ 7.4 TBps HBM
Peak Flops per chip	275 TFLOPS	459 TFLOPS	4614 TFLOPS

Peak Flops Per Watt (TDP)

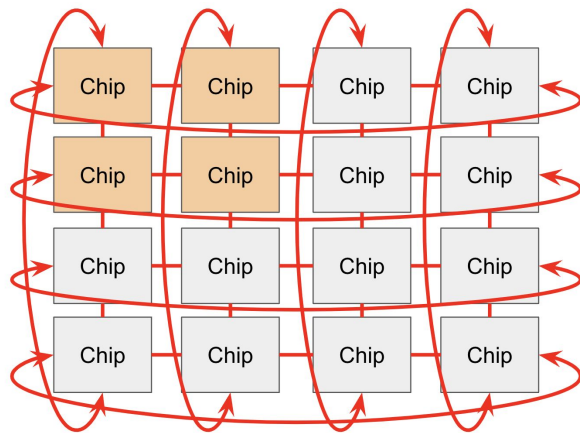


TPU Pod & Slice

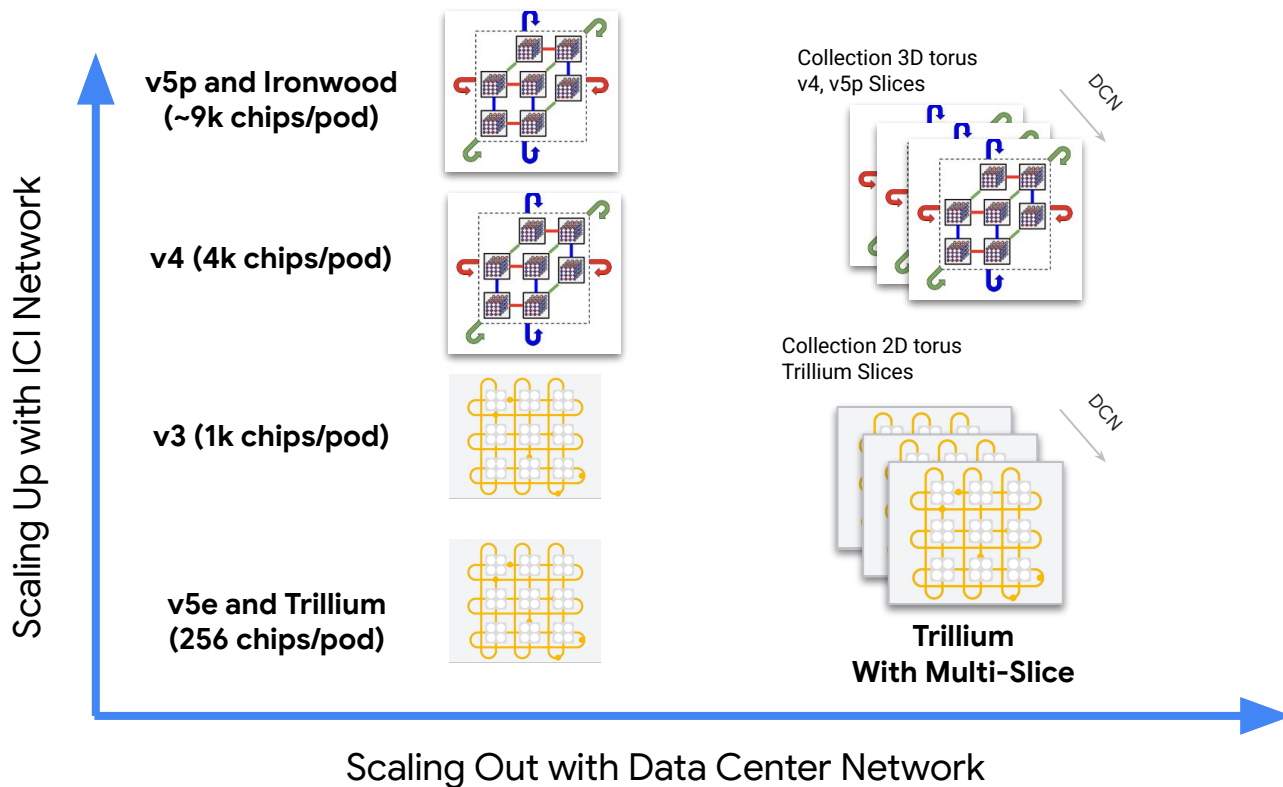
A TPU Pod is a single, unified package of multiple TPU chips interconnected via the Inter-Chip Interconnect (ICI). The number of TPU chips contained in a TPU Pod varies depending on the TPU version (TPU v6e: 256, TPU v7: 256/9,216).



Smaller slices can also be requested.



TPUs allow Google to massively scale up & out

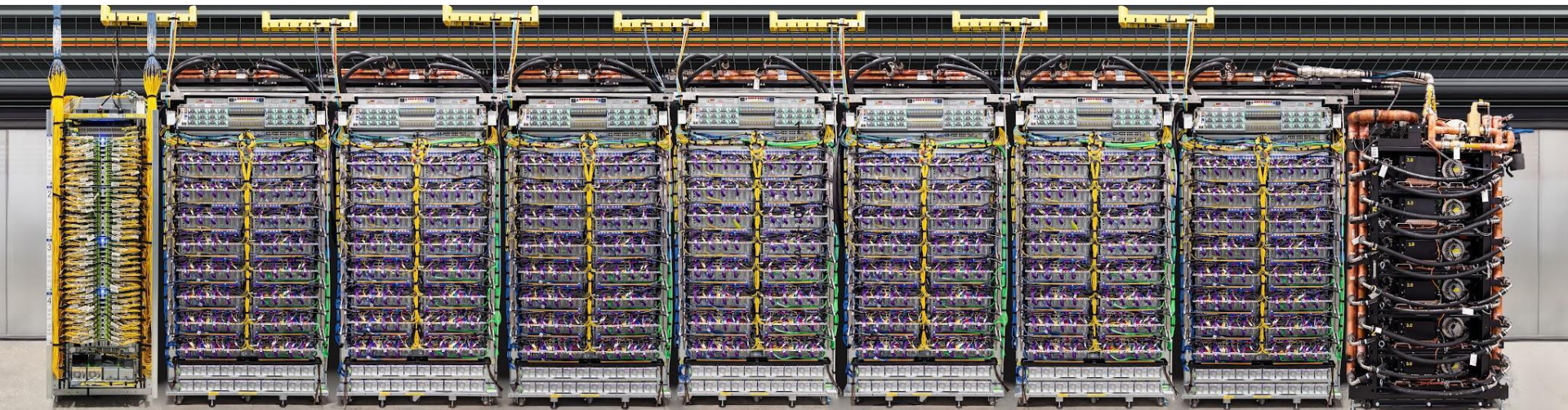


9,216

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.

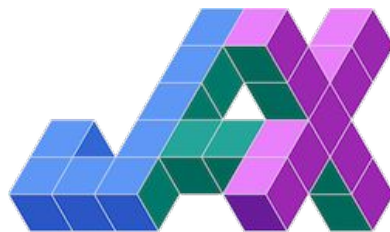
- **Core Philosophy:** Achieve Performance, Flexibility, and Scalability using **function transformations**.
- **Engine:** Uses the **XLA (Accelerated Linear Algebra)** compiler to generate highly optimized code for the target hardware.
- **Portability:** Enables running the same code, often without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

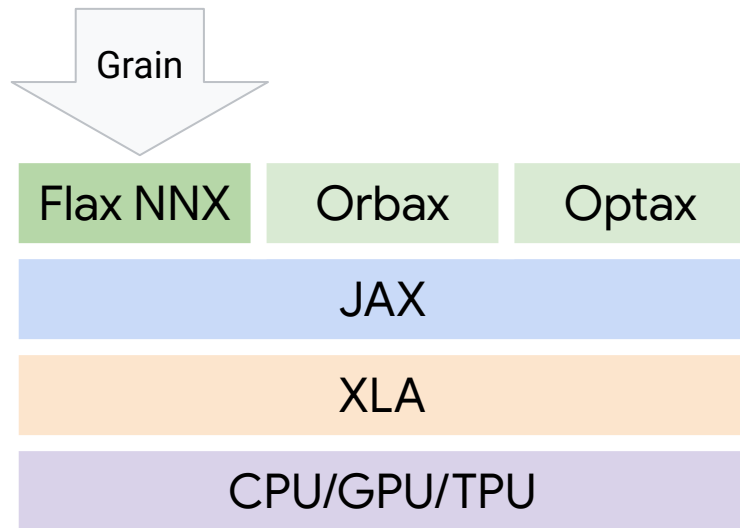


- You've seen results generated with JAX
- Google uses JAX for nearly all of its research and GenAI development
- Gemini, Gemma, Imagen, Veo, Waymo, etc. are all created using JAX

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.



These transformations can be arbitrarily composed, e.g.,

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

JAX Superpower: Flexible Parallelism

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

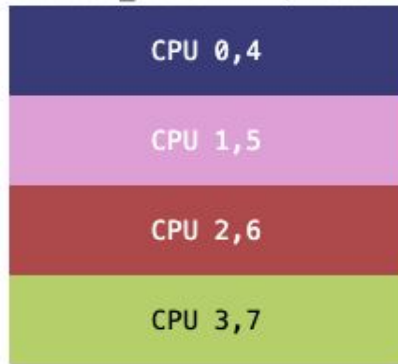
PyTorch: Library-Based

- 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.).

`sharded_model.w2 ('model', None)`

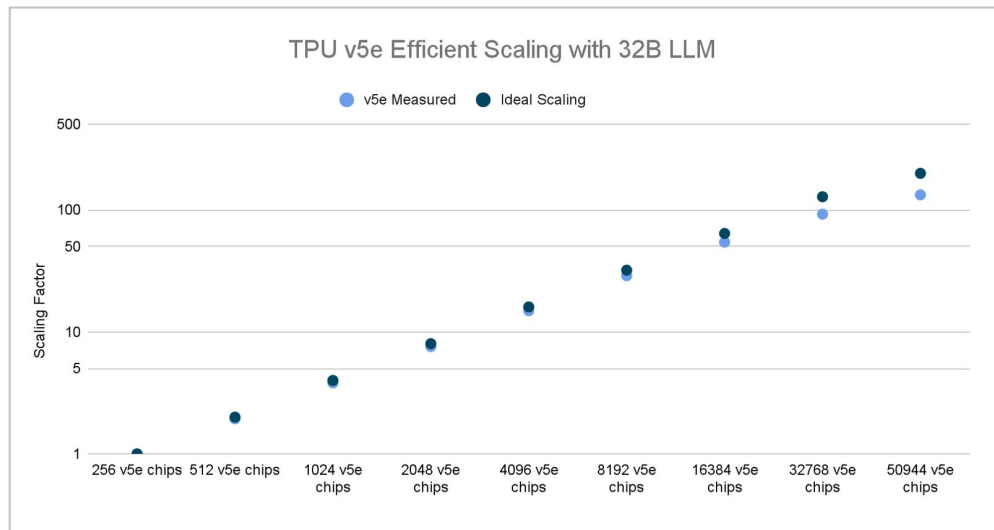


JAX Scalability: Scaling to 50,944 TPUs with JAX

JAX Scalability: TPUs

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

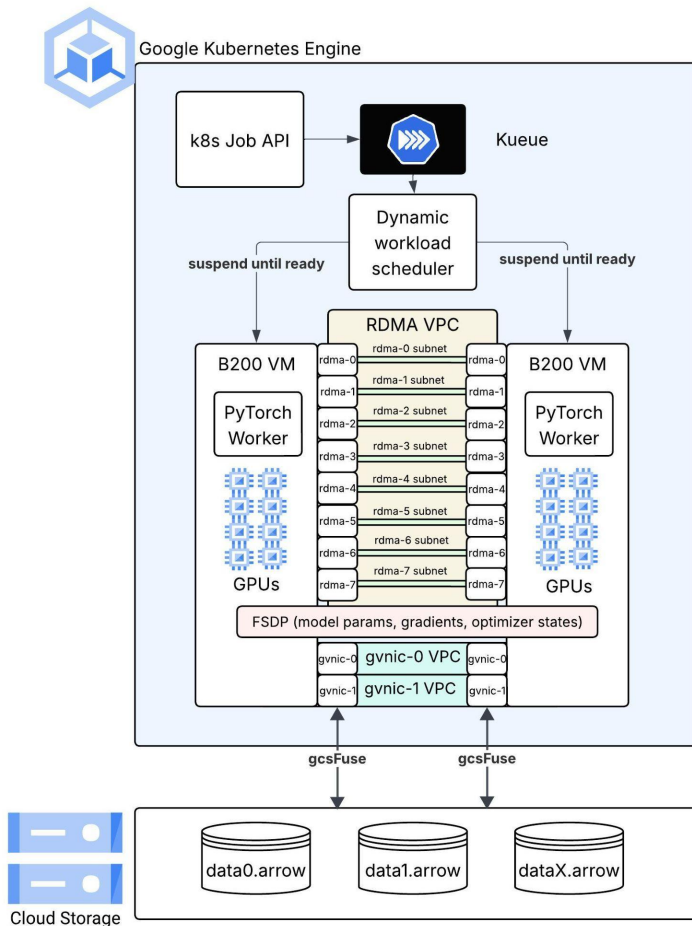
- 50,944 Cloud TPU v5e chips (spanning 199 Cloud TPU v5e pods)
- Near ideal scaling



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.

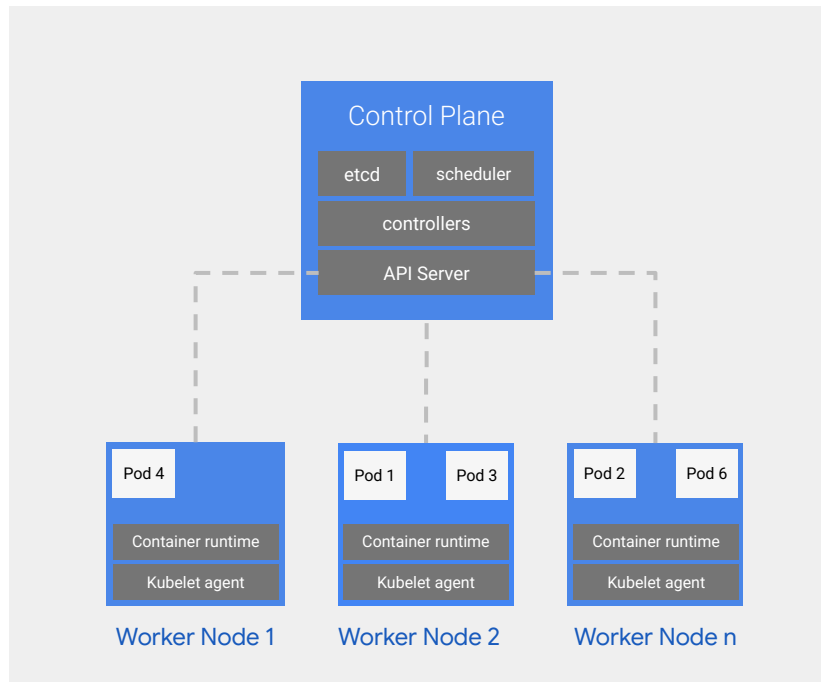
```
torchrun
--nnodes=$NUM_NODES
--nproc-per-node=$NUM_TRAINERS
--max-restarts=3
--rdzv-id=$JOB_ID
--rdzv-backend=c10d
--rdzv-endpoint=$HOST_NODE_ADDR
YOUR_TRAINING_SCRIPT.py (--arg1 ... train script args...)
```



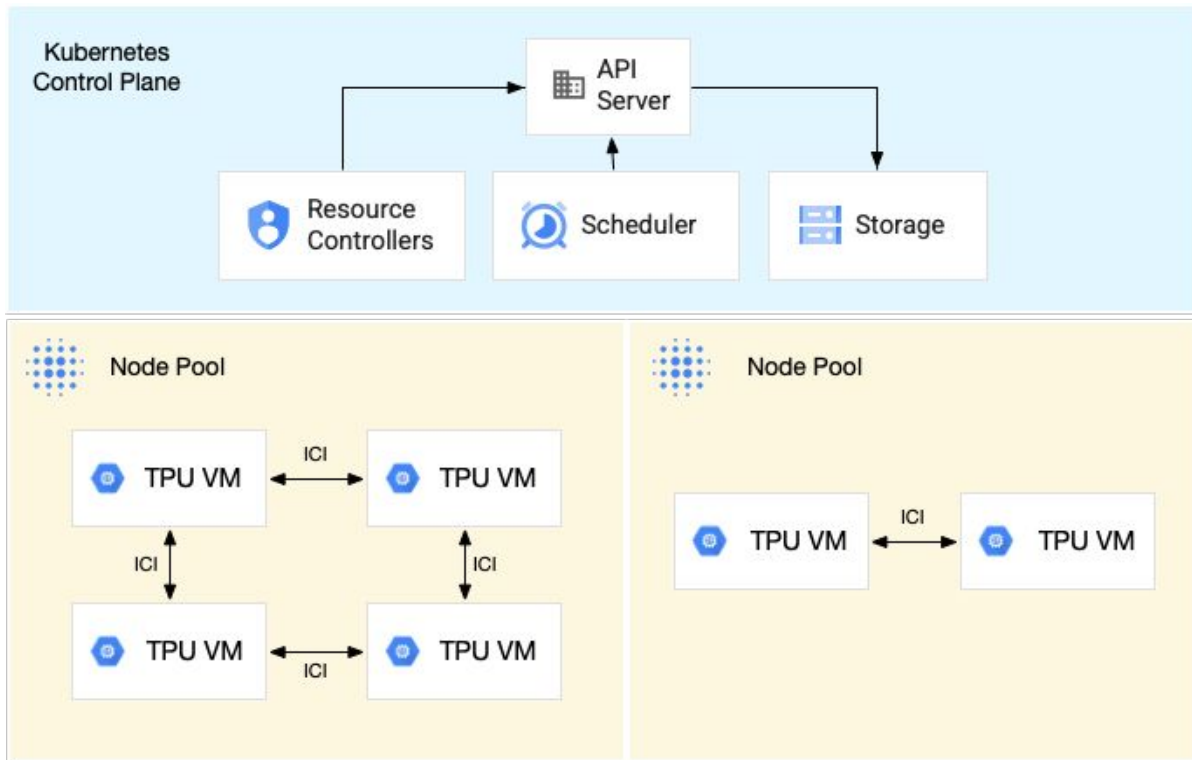
Kubernetes for Orchestration

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 [Terraform](#) 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: <https://github.com/GoogleCloudPlatform/cluster-toolkit>

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>