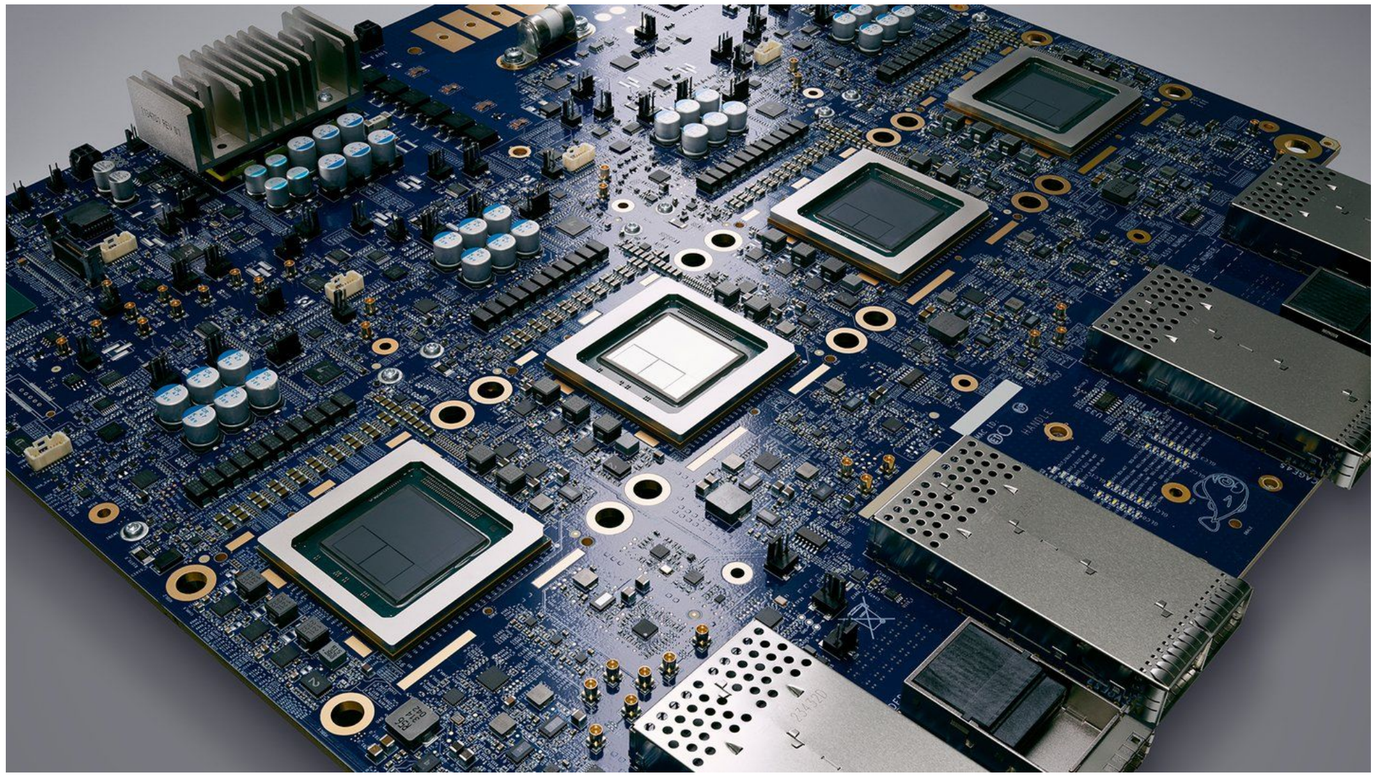
Intro to JAX & TPUs

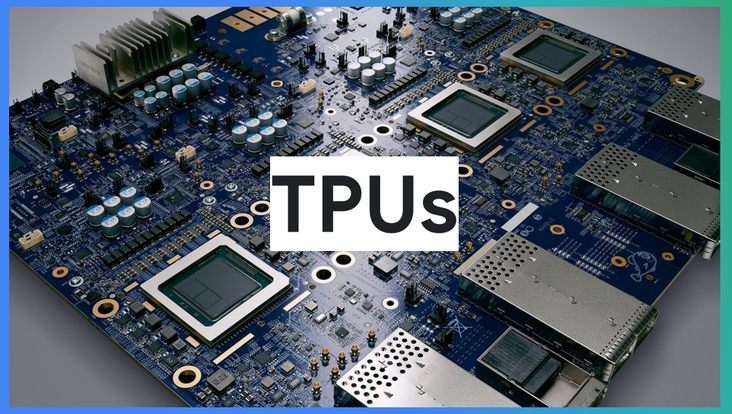


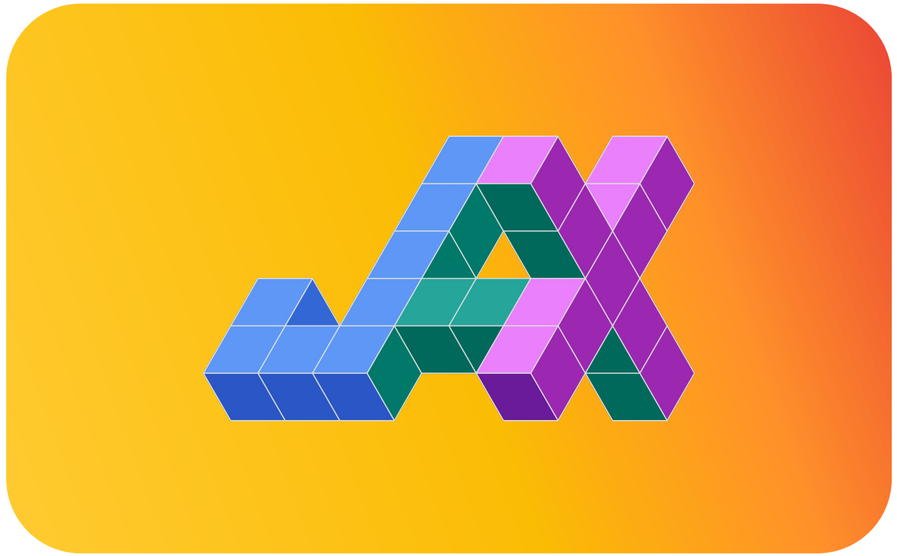
2025-11-06

Robert Dyro
@rdyro

Pallas
JAX kernel language

```
def add_kernel(x_ref, y_ref, out_ref):
  out_ref[:, :] = x_ref[...] + y_ref[...]
```

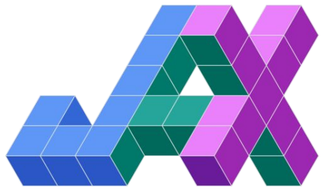




Familiar API

JAX provides a familiar NumPy-style API for ease of adoption by researchers and engineers.

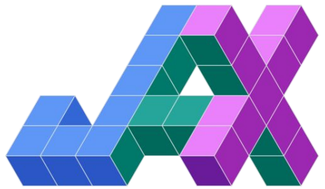
```
import jax.numpy as jnp


def predict(params, inputs):
  for W, b in params:
    outputs = jnp.dot(inputs, W) + b
    inputs = jnp.tanh(outputs)
  return outputs

def loss(params, batch):
  inputs, targets = batch
  preds = predict(params, inputs)
  return jnp.sum((preds - targets) ** 2)
```



JAX transformations

Composable function transformations for **automatic differentiation,** batching, compilation, and parallelization.

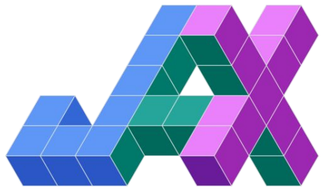
```
import jax.numpy as jnp
from jax import grad

def predict(params, inputs):
  for W, b in params:
    outputs = jnp.dot(inputs, W) + b
    inputs = jnp.tanh(outputs)
  return outputs

def loss(params, batch):
  inputs, targets = batch
  preds = predict(params, inputs)
  return jnp.sum((preds - targets) ** 2)

gradient_fun = grad(loss)
```



JAX transformations

Composable function transformations for automatic differentiation, **batching**, compilation, and parallelization.

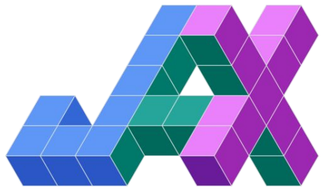
```
import jax.numpy as jnp
from jax import grad, vmap

def predict(params, inputs):
  for W, b in params:
    outputs = jnp.dot(inputs, W) + b
    inputs = jnp.tanh(outputs)
  return outputs

def loss(params, batch):
  inputs, targets = batch
  preds = predict(params, inputs)
  return jnp.sum((preds - targets) ** 2)

gradient_fun = grad(loss)
perexample_grads = vmap(
  grad(loss), in_axes=(None, 0))
```



JAX transformations

Composable function transformations for automatic differentiation, batching, **compilation**, and parallelization.

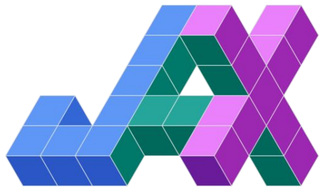
```
import jax.numpy as jnp
from jax import grad, vmap, jit

def predict(params, inputs):
  for W, b in params:
    outputs = jnp.dot(inputs, W) + b
    inputs = jnp.tanh(outputs)
  return outputs

def loss(params, batch):
  inputs, targets = batch
  preds = predict(params, inputs)
  return jnp.sum((preds - targets) ** 2)

gradient_fun = jit(grad(loss))
perexample_grads = jit(
    vmap(grad(loss), in_axes=(None, 0)))
```



JAX transformations

Composable function transformations for automatic differentiation, batching, compilation, and **parallelization**.

```
import jax.numpy as jnp
from jax import grad, vmap, jit

def predict(params, inputs):
  for W, b in params:
    outputs = jnp.dot(inputs, W) + b
    inputs = jnp.tanh(outputs)
  return outputs

def loss(params, batch):
  inputs, targets = batch
  preds = predict(params, inputs)
  return jnp.sum((preds - targets) ** 2)

gradient_fun = jit(grad(loss))
perexample_grads = jit(
    vmap(grad(loss), in_axes=(None, 0)))
```

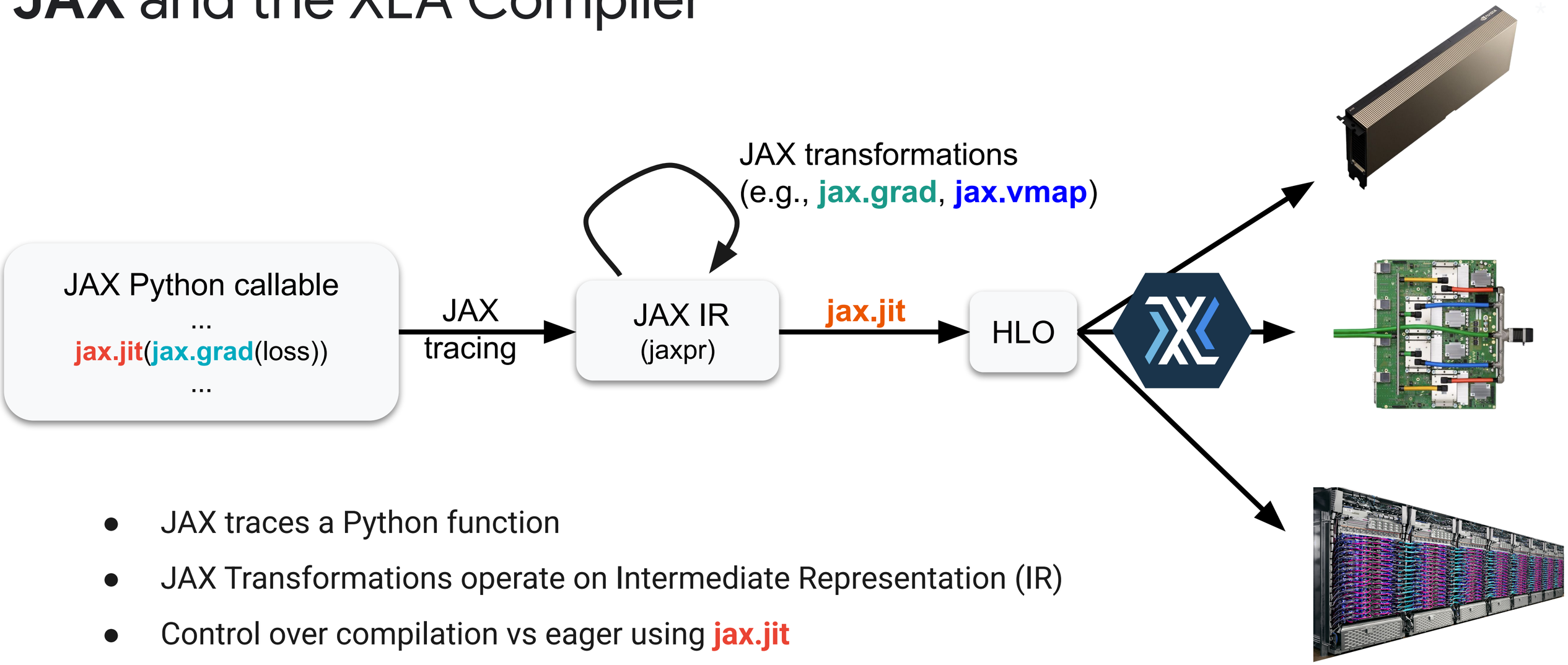


JAX transformations

Composable function transformations for automatic differentiation, batching, compilation, and **parallelization**.

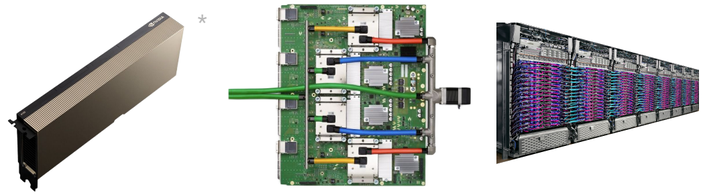
```
import jax.numpy as jnp
from jax import grad, vmap, jit

def predict(params, inputs):
  for W, b in params:
    outputs = jnp.dot(inputs, W) + b
    inputs = jnp.tanh(outputs)
  return outputs

def loss(params, batch):
  inputs, targets = batch
  preds = predict(params, inputs)
  return jnp.sum((preds - targets) ** 2)

gradient_fun = jit(grad(loss))
perexample_grads = jit(
    vmap(grad(loss), in_axes=(None, 0)),
    in_shardings=..., out_shardings=...)
```

JAX and the XLA Compiler



- JAX traces a Python function
- JAX Transformations operate on Intermediate Representation (IR)
- Control over compilation vs eager using **jax.jit**
- Same JAX code runs on CPU/GPU/TPU/pods

* Image: NVIDIA



Run anywhere and scale automatically

Any hardware: CPU, GPU (NVIDIA & AMD),
TPU Unified API: Any scale!

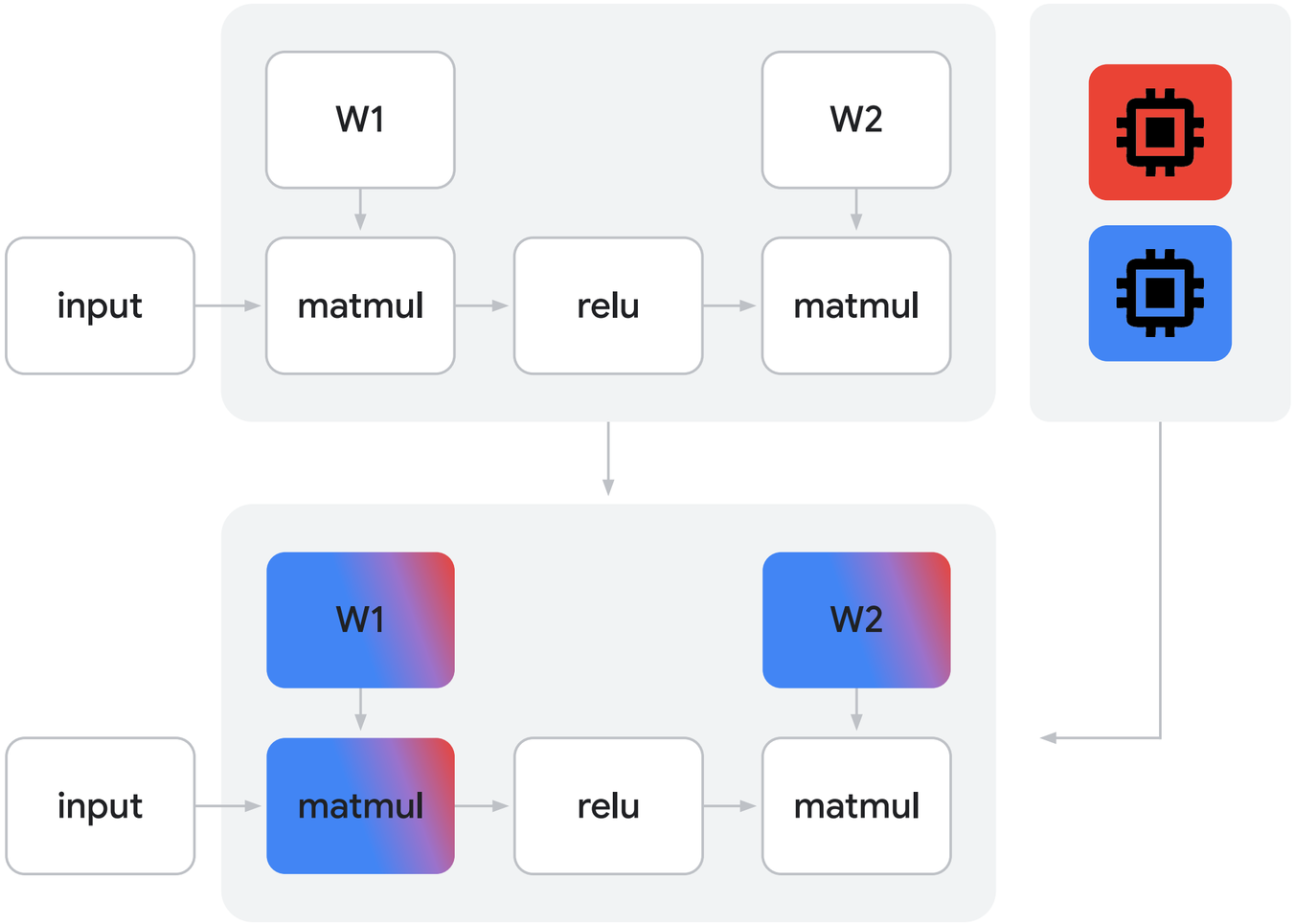
* Image: NVIDIA

Unified parallel computing with array sharding

jit – unified global view of the program
data parallelism / model parallelism / pipeline parallelism

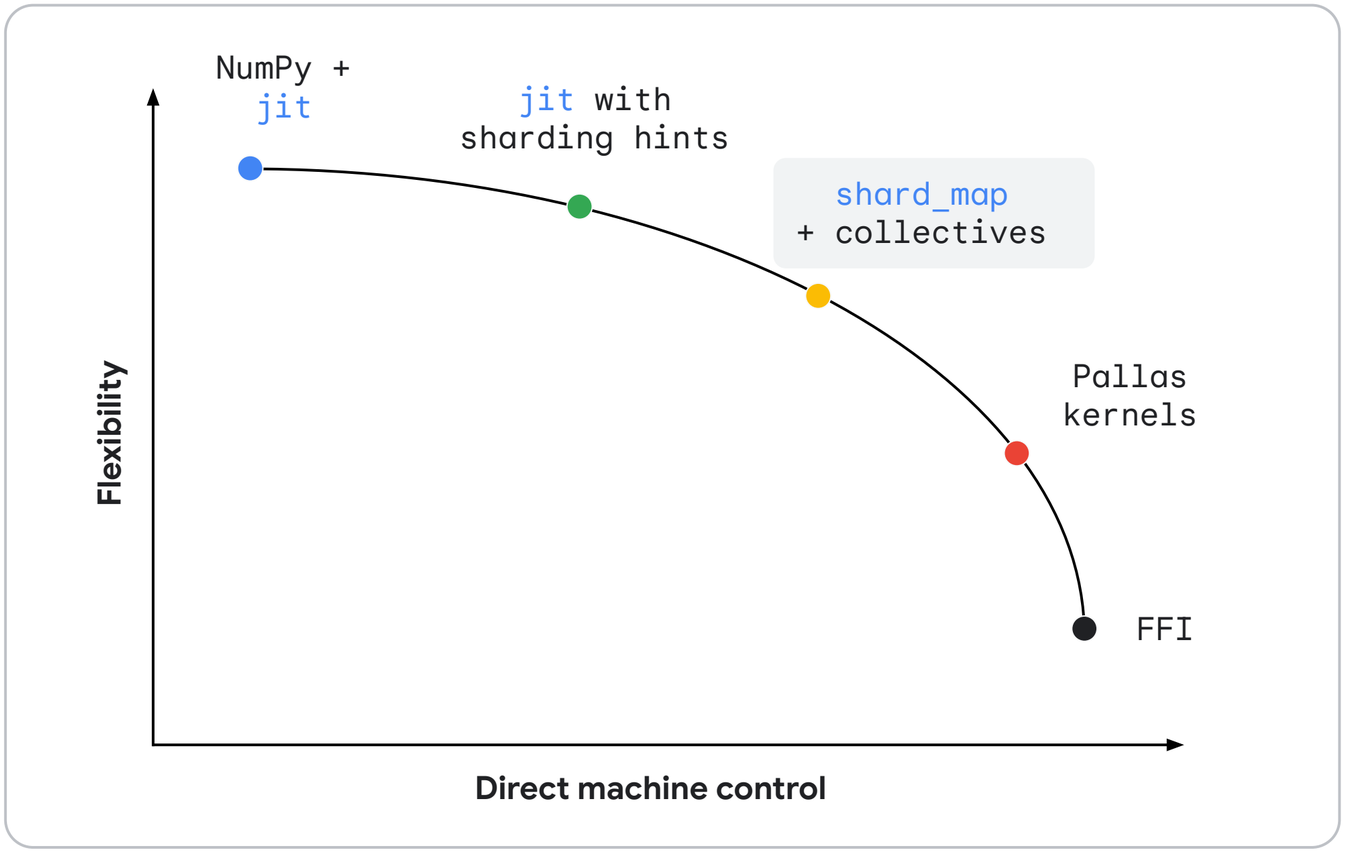
portable
CPU / GPU / TPU / ...
old, current & future hardware

scalable
1 chip ⟶ 10 chips ⟶ 10,000 chips



JAX: The escape hatch hierarchy

Flexibility vs. control



Math

- Compiler, take the wheel!
- Here's a hint
- I'll handle comms
- Kernel languages
- FFI

Physics

JAX: `shard_map` per-device view

- drop to per-device view
- control communication explicitly
- composable
  - e.g., full autodiff support
  - e.g., automatic batching

```
import jax.numpy as jnp


@partial(jax.shard_map, mesh=mesh,
    in_specs=(P('x', 'y'), P('y', None)),
    out_specs=P('x', None))
def matmul_basic(a_block, b_block):
  # a_block: f32[2, 8]
  # b_block: f32[8, 4]

  # compute
  z_partialsum = jnp.dot(a_block, b_block)

  # communicate
  z_block = jax.lax.psum(z_partialsum, 'y')

  # c_block: f32[2, 4]
  return z_block
```

`shard_map`
explicit collectives

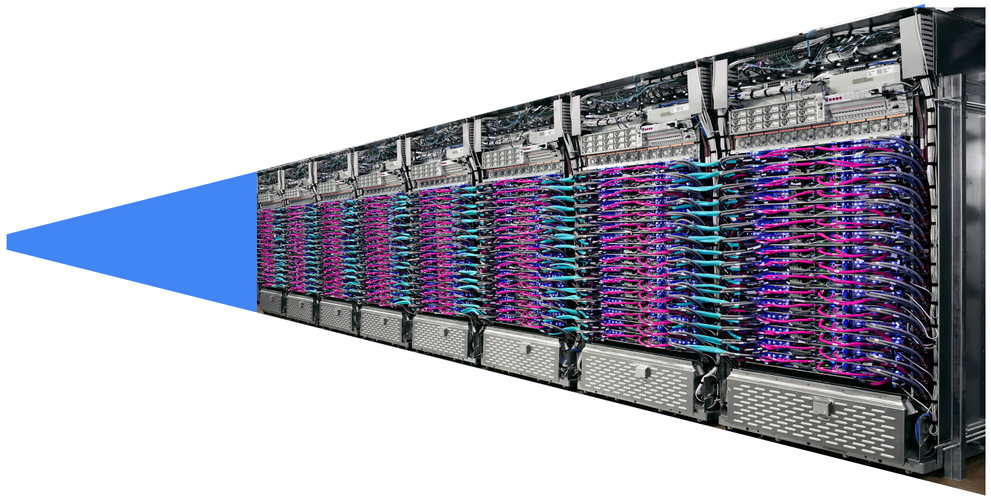
- Mixture-of-Experts models (MoE)
- Custom communication & compute overlap
- Custom scientific distributed computing

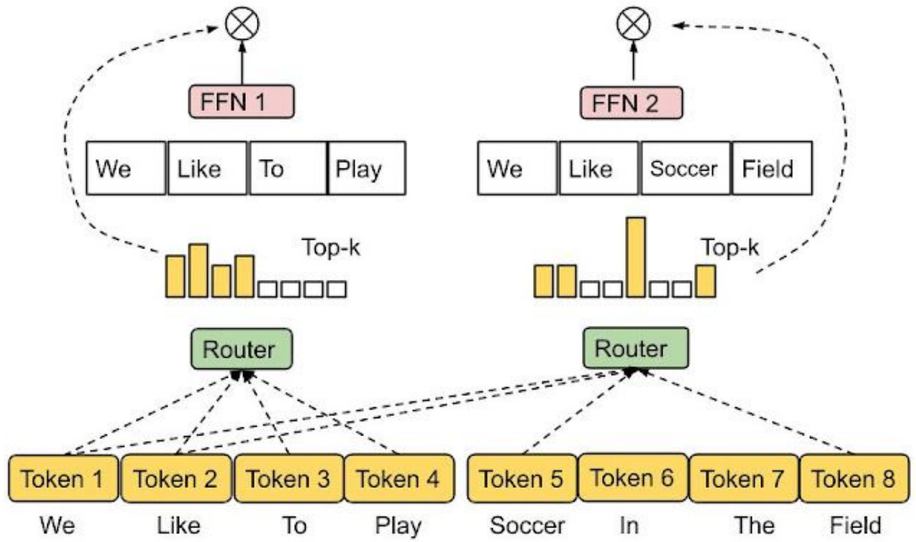
all-reduce
`jax.lax.psum`

all-gather
`jax.lax.all_gather`

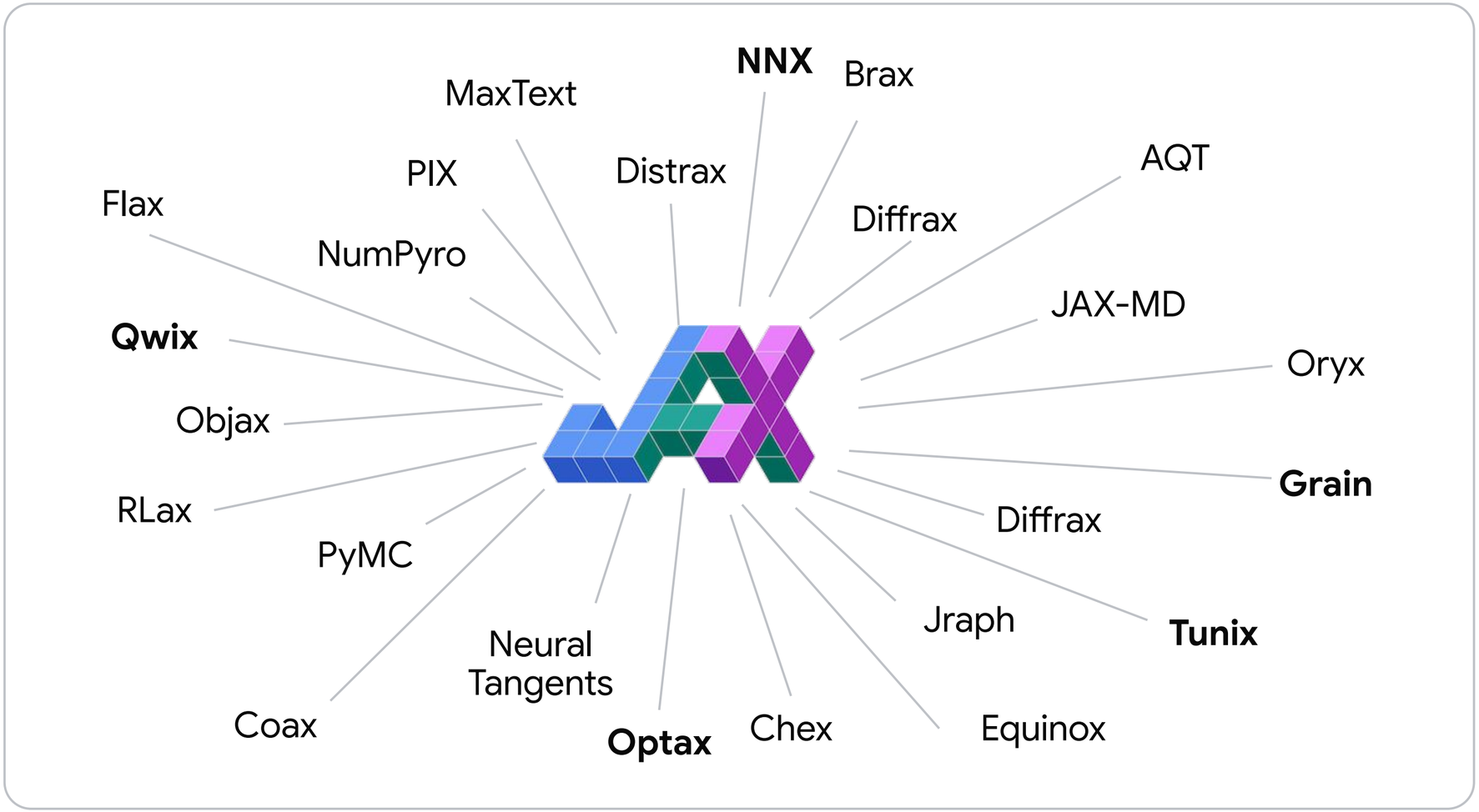
all-to-all
`jax.lax.all_to_all`

permute
`jax.lax.ppermute`





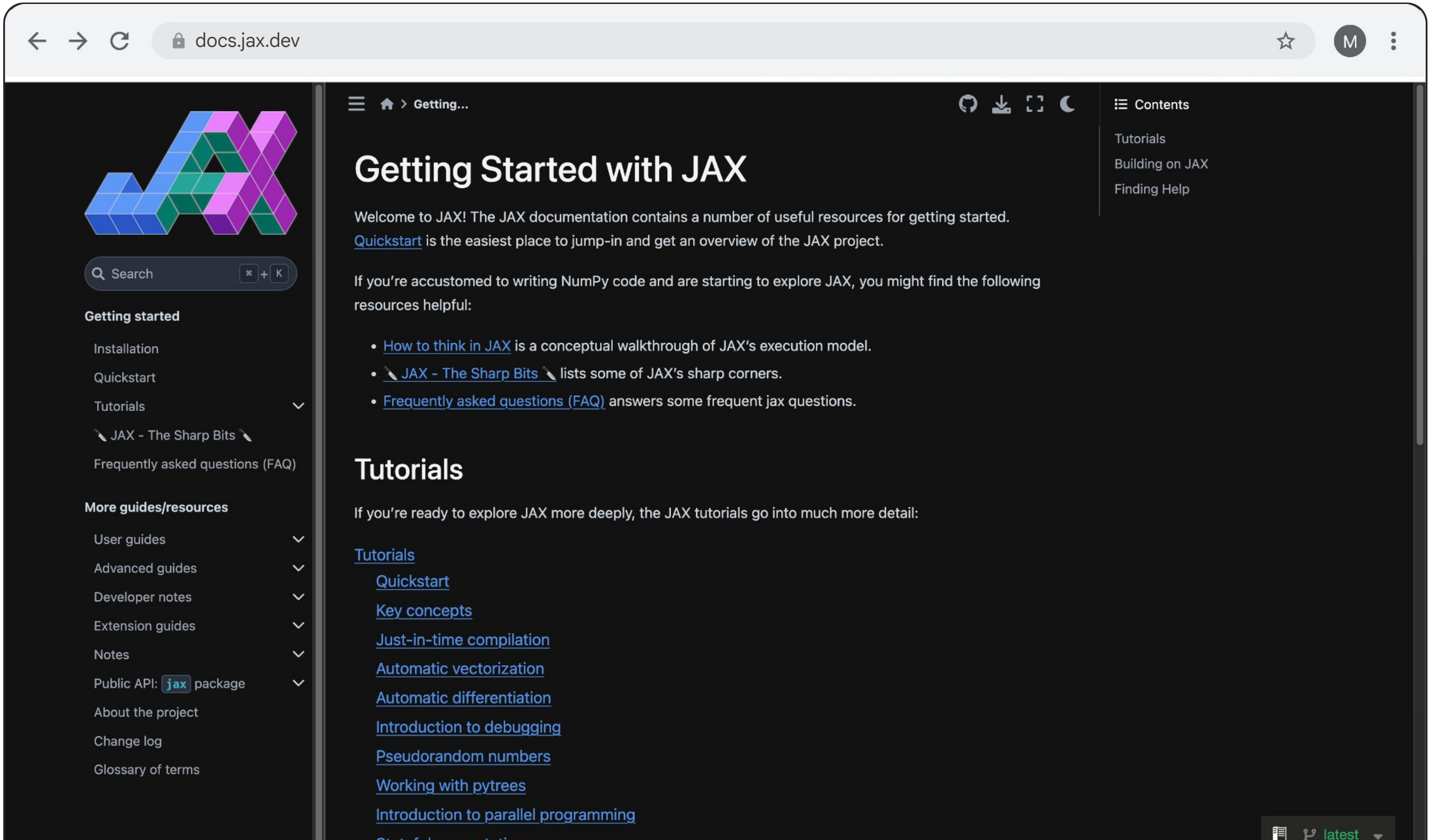
JAX Ecosystem

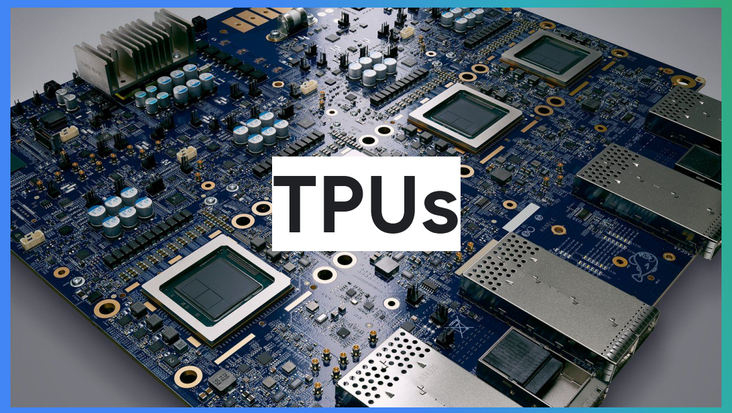


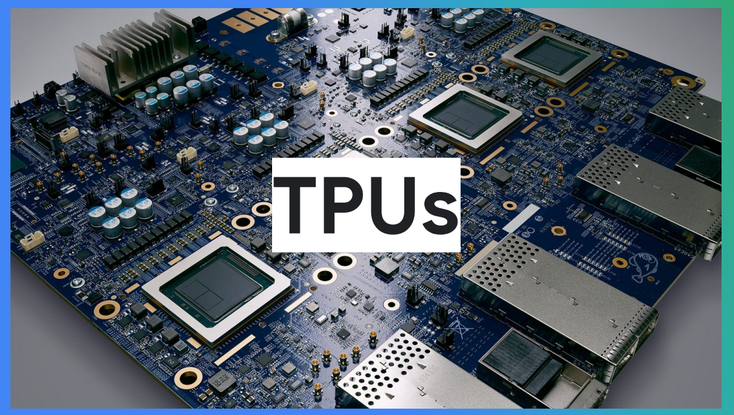
NNX: Object Oriented Neural Nets

Optax: Fast Optimizers in JAX

Grain: Feeding JAX Models


docs.jax.dev
M
Search ⌘ + K
Getting started
Installation
Quickstart
Tutorials
🔪 JAX - The Sharp Bits 🔪
Frequently asked questions (FAQ)
More guides/resources
User guides
Advanced guides
Developer notes
Extension guides
Notes
Public API: jax package
About the project
Change log
Glossary of terms
Getting...
Contents
Tutorials
Building on JAX
Finding Help
Getting Started with JAX
Welcome to JAX! The JAX documentation contains a number of useful resources for getting started. Quickstart is the easiest place to jump-in and get an overview of the JAX project.
If you're accustomed to writing NumPy code and are starting to explore JAX, you might find the following resources helpful:
How to think in JAX is a conceptual walkthrough of JAX's execution model.
🔪 JAX - The Sharp Bits 🔪 lists some of JAX's sharp corners.
Frequently asked questions (FAQ) answers some frequent jax questions.
Tutorials
If you're ready to explore JAX more deeply, the JAX tutorials go into much more detail:
Tutorials
Quickstart
Key concepts
Just-in-time compilation
Automatic vectorization
Automatic differentiation
Introduction to debugging
Pseudorandom numbers
Working with pytrees
Introduction to parallel programming
latest


TPUs



Super-fast
Compute
A✖B

Scalable
Chip
Networking

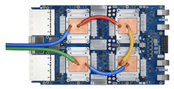
Uncomplicated
Architecture
Easy Low-level
Control

Timeline

2015

TPU v1

Internal inference accelerator



2020

TPU v3

Liquid cooled

2023

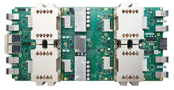
TPU v5e

Purpose-built for medium and large-scale training and inference

2024

Trillium

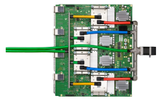
Exceptional performance and efficiency



2018

TPU v2

256 chips distributed shared memory



2022

TPU v4

Optically reconfigurable, 3D Torus architecture

2023

TPU v5p

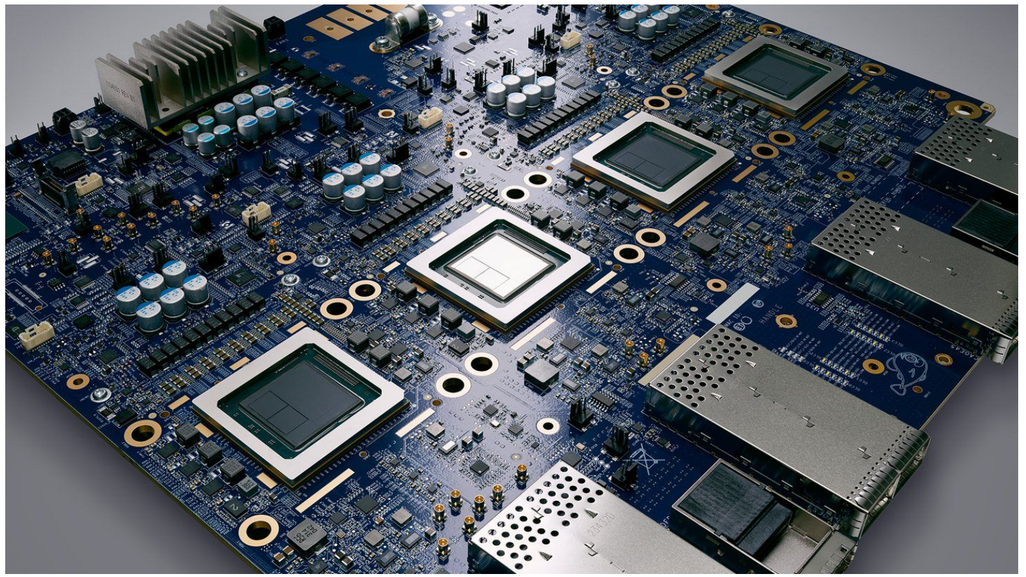
Powerful, scalable, and flexible AI accelerator

2025

Ironwood

5x peak compute
6x more HBM
2x more efficient*

An example: TPU v6e - Trillium



Specification	v6e
Peak compute per chip (bf16)	918 TFLOPs
Peak compute per chip (Int8)	1836 TOPs
HBM capacity per chip	32 GB
HBM bandwidth per chip	1600 GBps
Inter-chip interconnect (ICI) bandwidth	3200 Gbps
ICI ports per chip	4
TPU Pod size	256 chips
Interconnect topology	2D torus
BF16 peak compute per Pod	234.9 PFLOPs
All-reduce bandwidth per Pod	102.4 TB/s
Special features	SparseCore

MXU FLOPS

HBM per chip

ICI for 2D slices with wraparound connections

ICI limit, DCN after that (multiple pods)

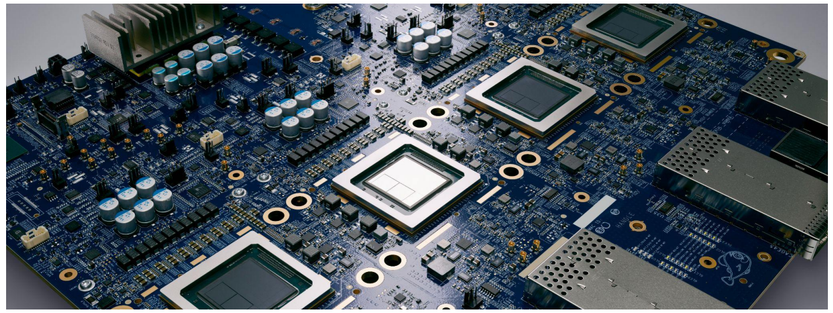
gather/scatter + async collectives

https://cloud.google.com/tpu/docs/v6e

Even Faster: **Ironwood** (7th Gen TPU) available now

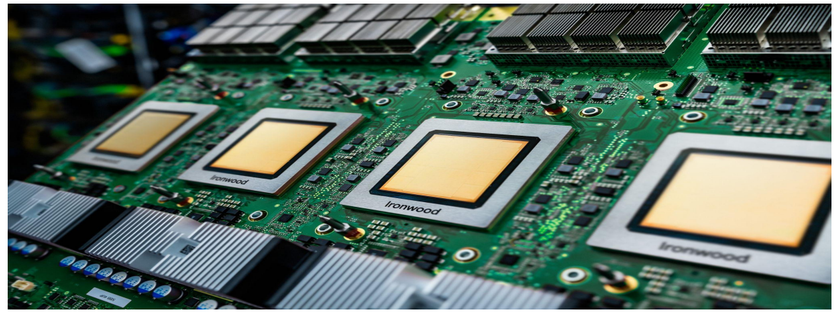
v5e, v5p

- 3x faster training vs v4
- 4x FLOPS/pod vs v4
- 8960 (v5p) or 256 chips (v5e) per pod



Trillium

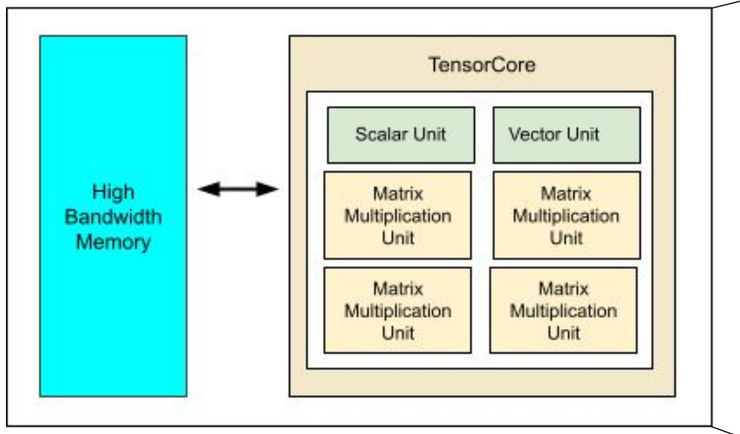
- 4x training perf vs v5e
- 5x inference perf vs v5e
- 256 chips per pod

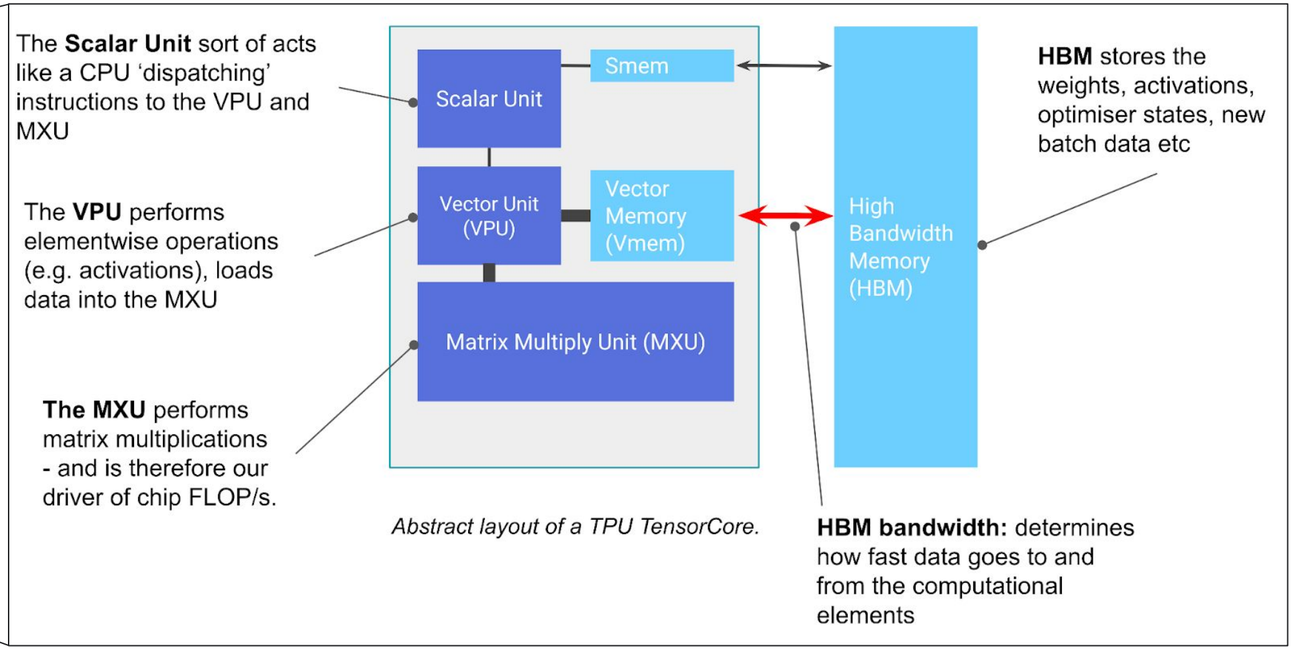


Ironwood

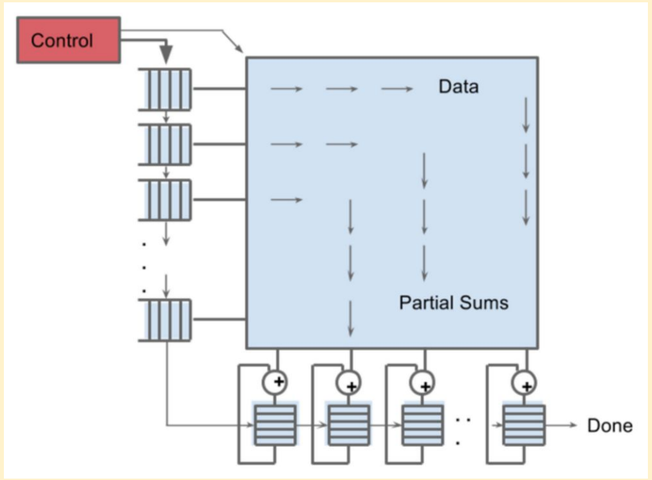
- 5x peak compute vs Trillium
- 6x more HBM vs Trillium
- 2.8x more power-efficient vs v5p
- 9,216 (v7x) or 256 (v7) chips per pod

How a TPU works?





MXU

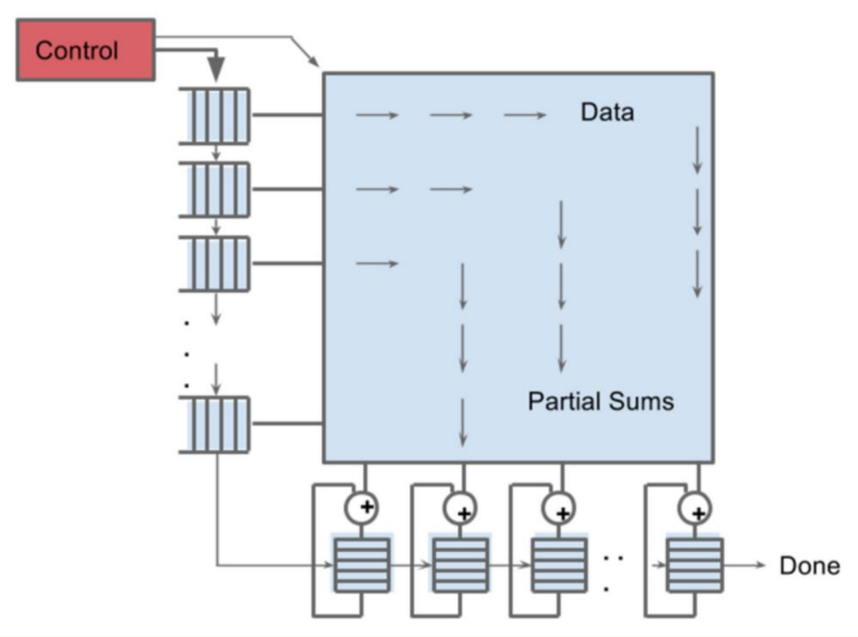


Super-fast
Compute
A✖B

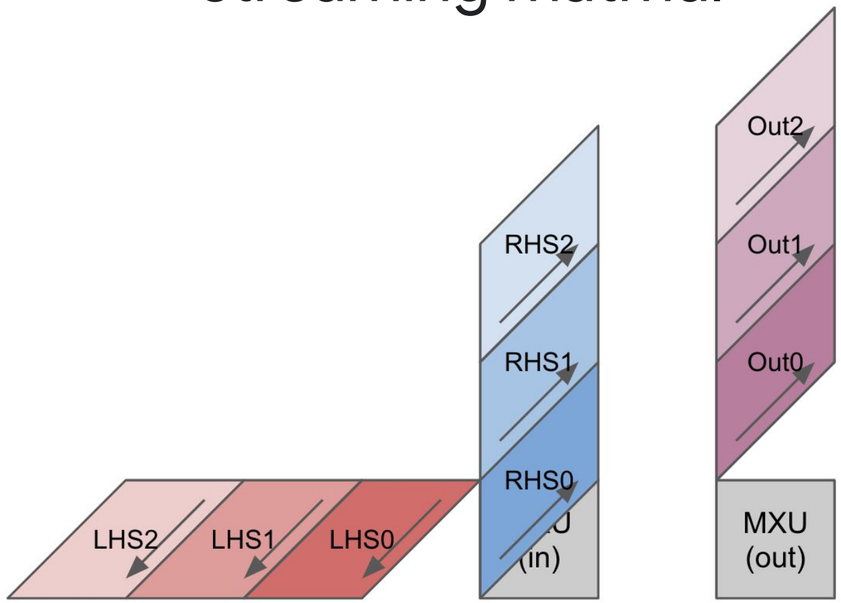
How to Scale Your Model: https://jax-ml.github.io/scaling-book/tpus/

How a TPU works?
The Systolic Array

MXU
Systolic Array

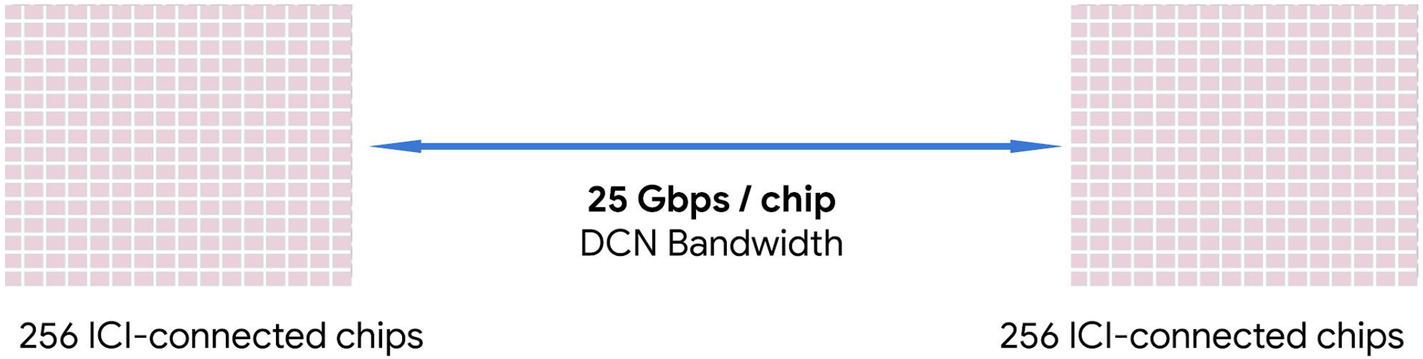


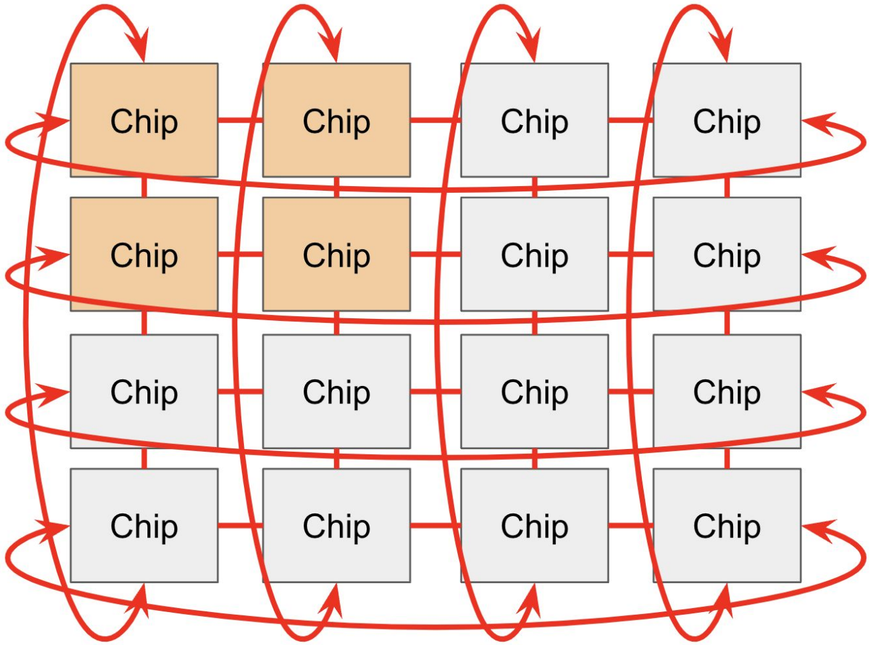
Systolic array:
streaming matmul



How to Scale Your Model: https://jax-ml.github.io/scaling-book/tpus/

Inter-chip Networking

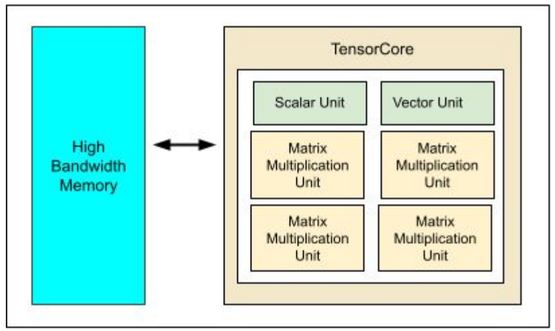




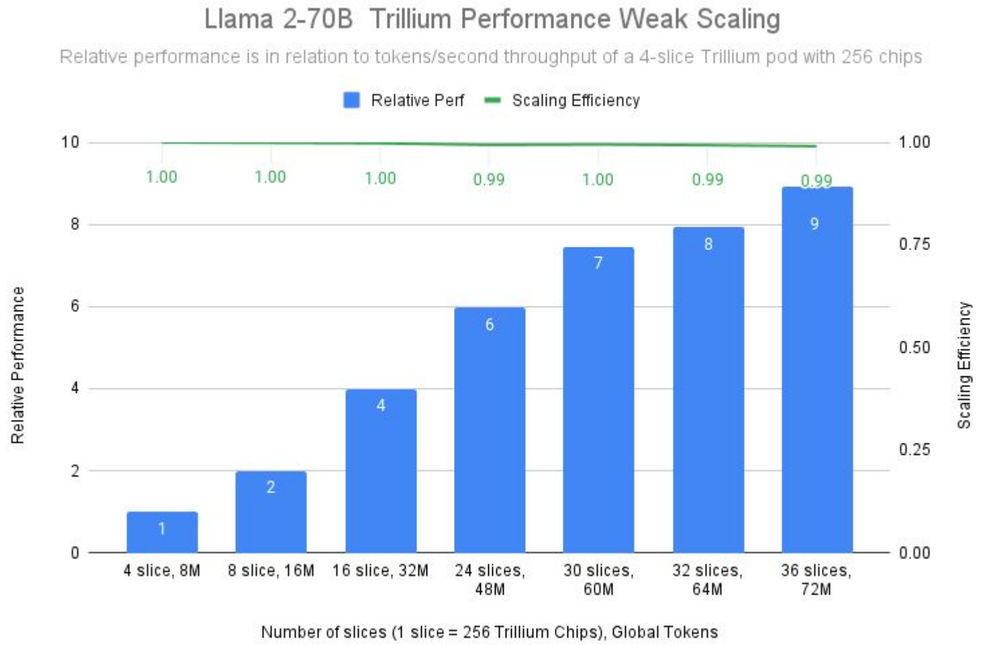
Scalable
Chip
Networking

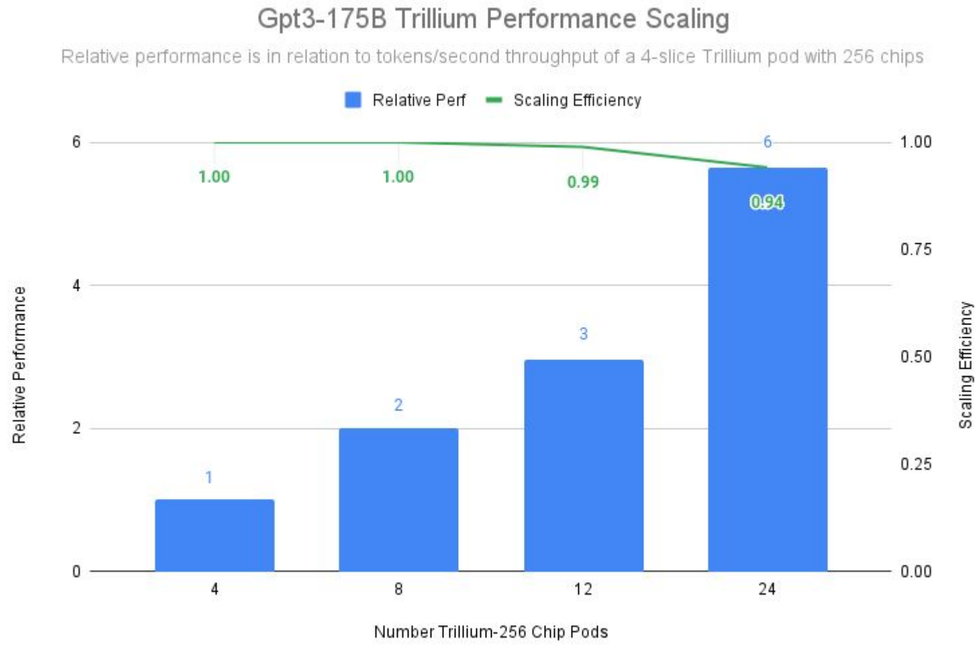
https://insidehpc.com/2019/05/google-cloud-tpu-pods-speed-machine-learning/

Near-optimal Scaling



Model	Pod size	Host size	HBM capacity/chip	HBM BW/chip (bytes/s)	FLOPs/s/chip (bf16)	FLOPs/s/chip (int8)
TPU v3	32x32	4x2	32GB	9.0e11	1.4e14	1.4e14
TPU v4p	16x16x16	2x2x1	32GB	1.2e12	2.75e14	2.75e14
TPU v5p	16x20x28	2x2x1	96GB	2.8e12	4.59e14	9.18e14
TPU v5e	16x16	4x2	16GB	8.1e11	1.97e14	3.94e14
TPU v6e	16x16	4x2	32GB	1.6e12	9.20e14	1.84e15



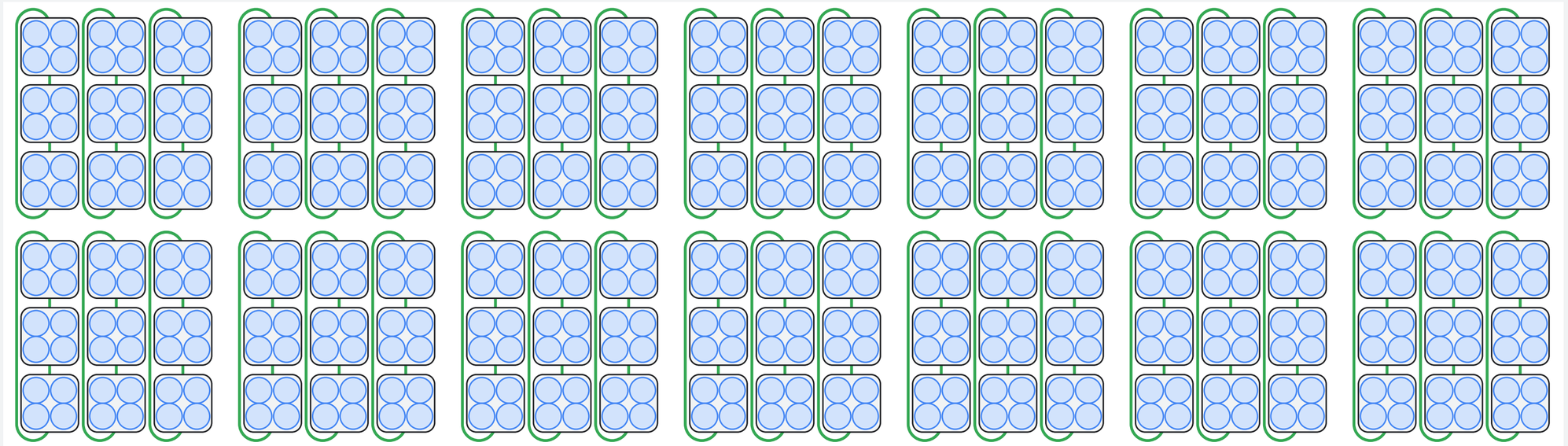


Ironwood (7th gen TPU) pods get huge!

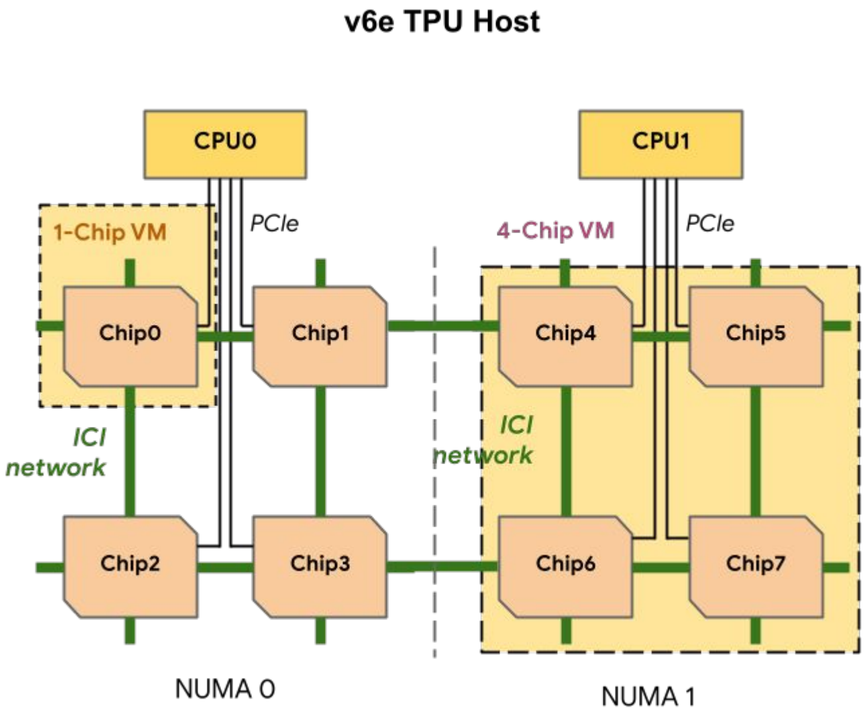
TPU superpod (ICI domain)

9,216 TPUs and 1.77PB of shared memory

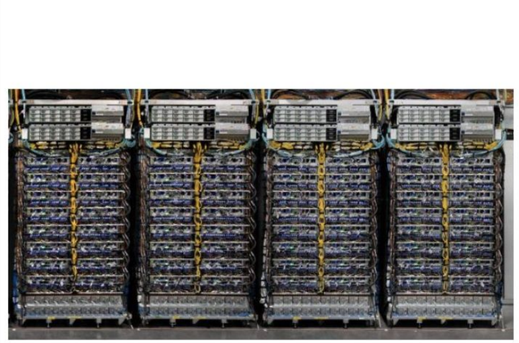
Connect dozens of these via Data Center Network (DCN)



Provisioning a TPU

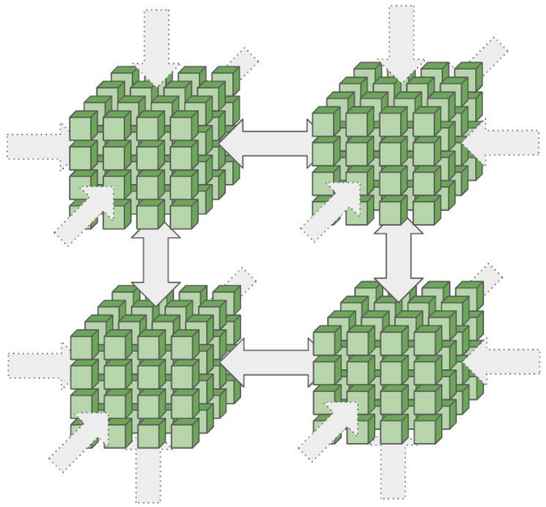


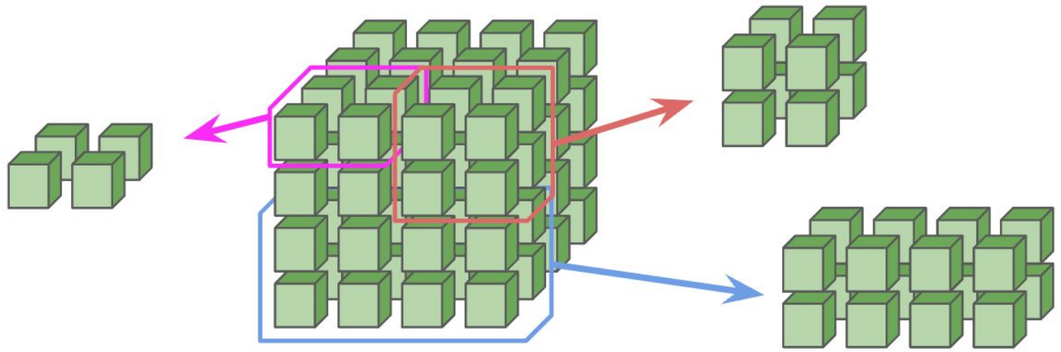
- powerful multi-core server CPU
- lots of CPU RAM (700+ GB)
- efficient network storage 10+ Gbps



Each rack corresponds to 4x4x4 TPUv4s

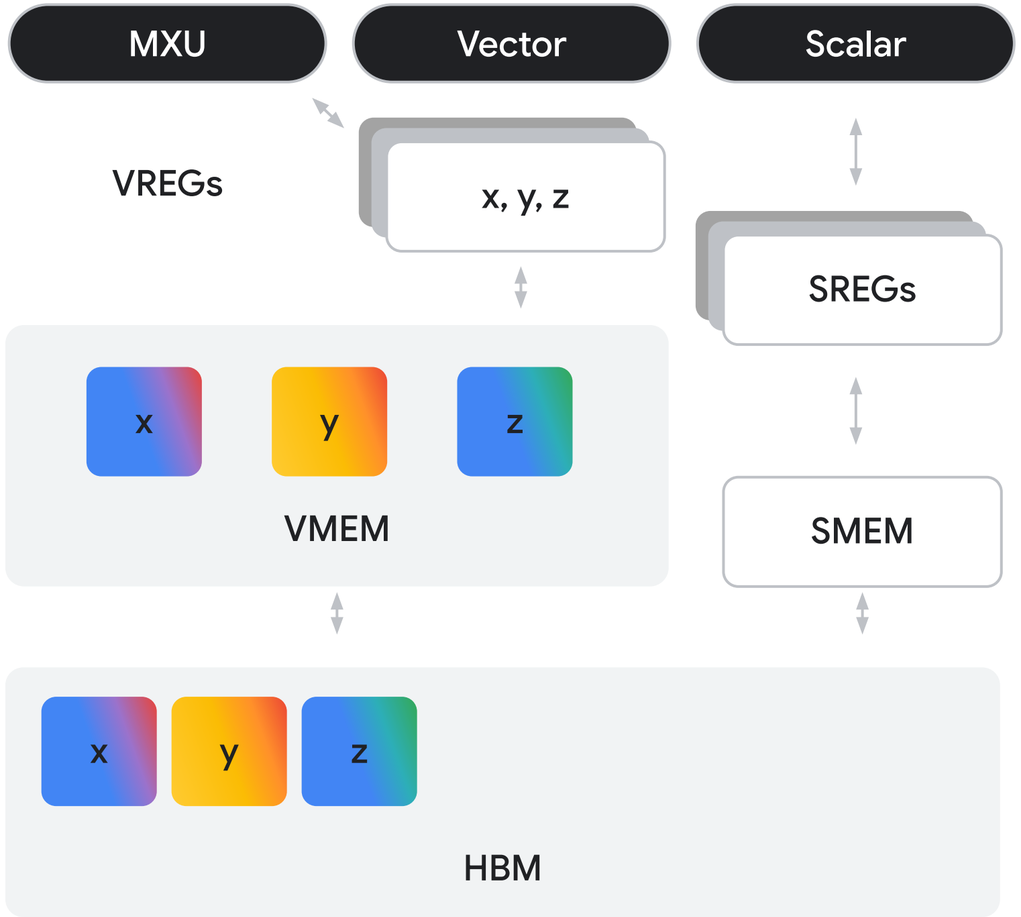
These can be configured into arbitrary 4ix4jx4k toruses via optical interconnects



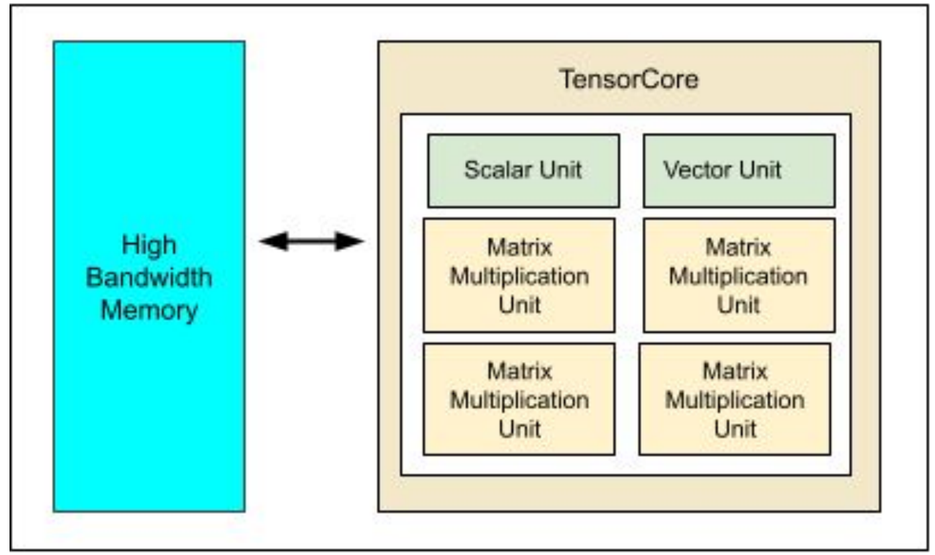


Smaller slices can also be requested.

TPU Memory pipeline



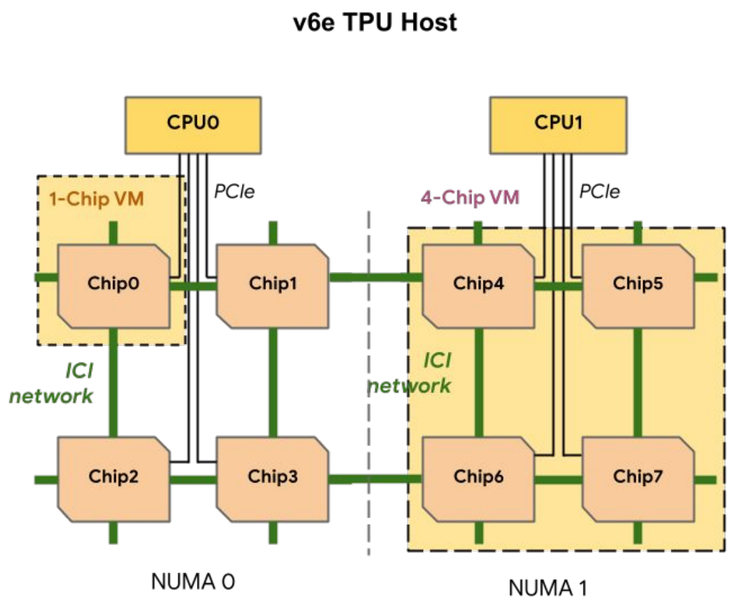
Uncomplicated Architecture
Easy Low-level Control



TPUs

Super-fast
Compute
A✖B

Specification	v6e
Peak compute per chip (bf16)	918 TFLOPs
Peak compute per chip (Int8)	1836 TOPs
HBM capacity per chip	32 GB
HBM bandwidth per chip	1600 GBps
Inter-chip interconnect (ICI) bandwidth	3200 Gbps
Interconnect topology	2D torus



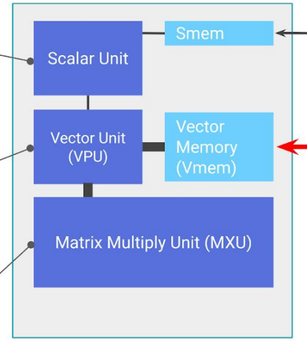
Scalable
Chip
Networking

Uncomplicated
Architecture
Easy Low-level
Control

The **Scalar Unit** sort of acts like a CPU 'dispatching' instructions to the VPU and MXU

The **VPU** performs elementwise operations (e.g. activations), loads data into the MXU

The MXU performs matrix multiplications - and is therefore our driver of chip FLOP/s.



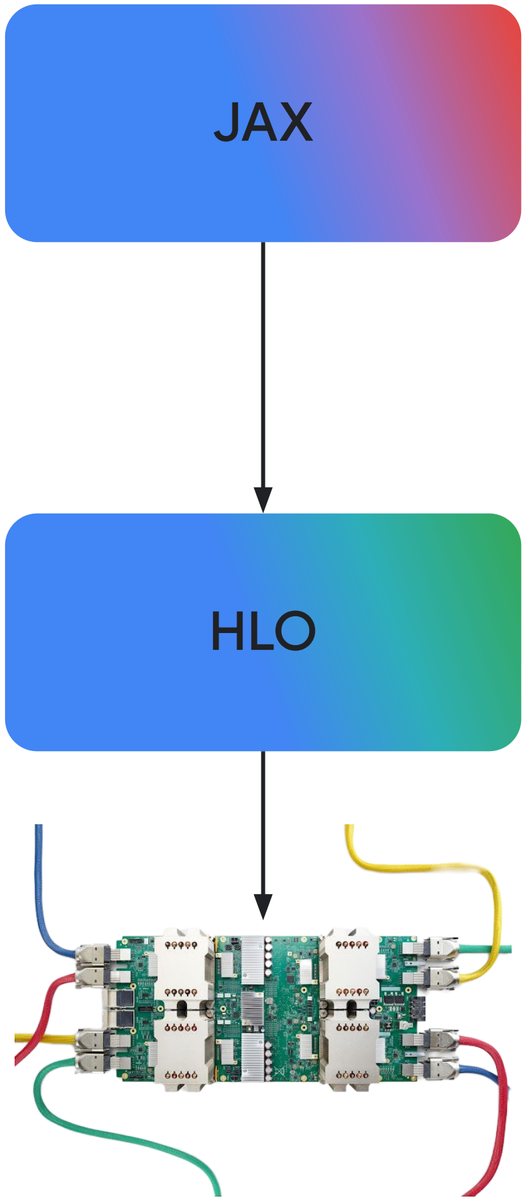
Abstract layout of a TPU TensorCore.

HBM stores the weights, activations, optimiser states, new batch data etc

HBM bandwidth: determines how fast data goes to and from the computational elements

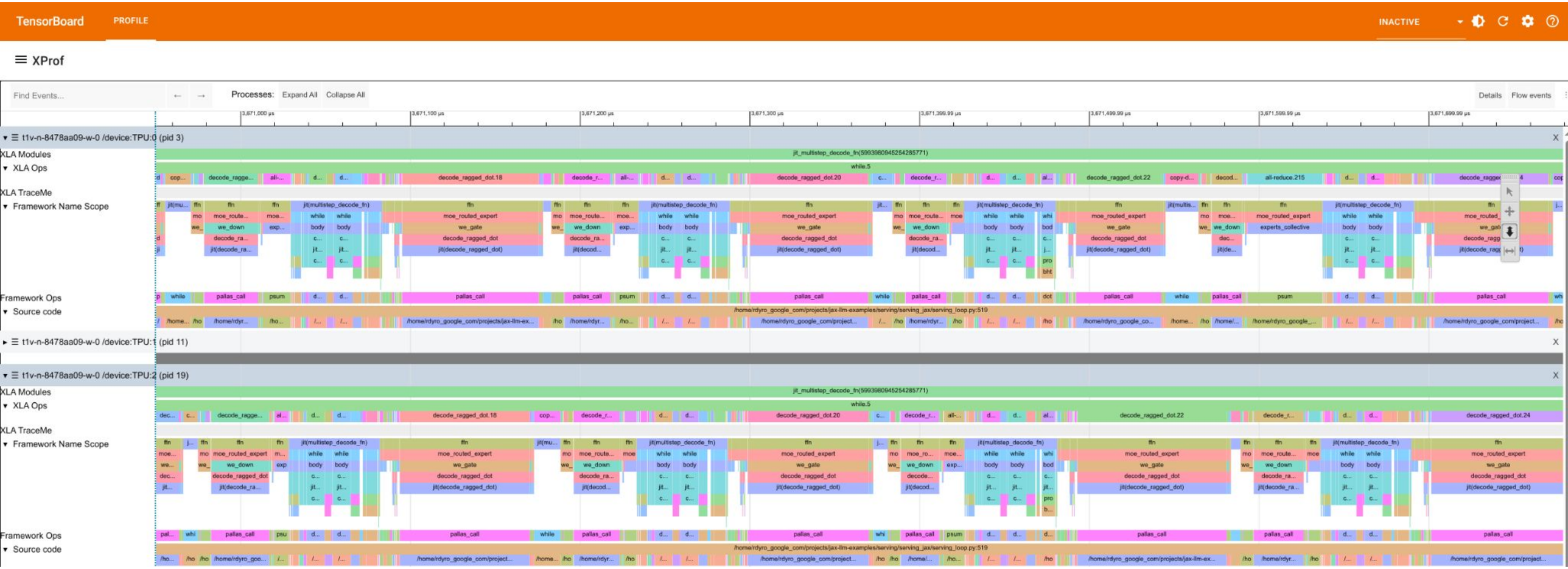
Programming TPUs

Programming TPUs



(not just JAX, thanks to XLA)

TPU Performance:
Amazing compute, fast collectives and state of the art compiler

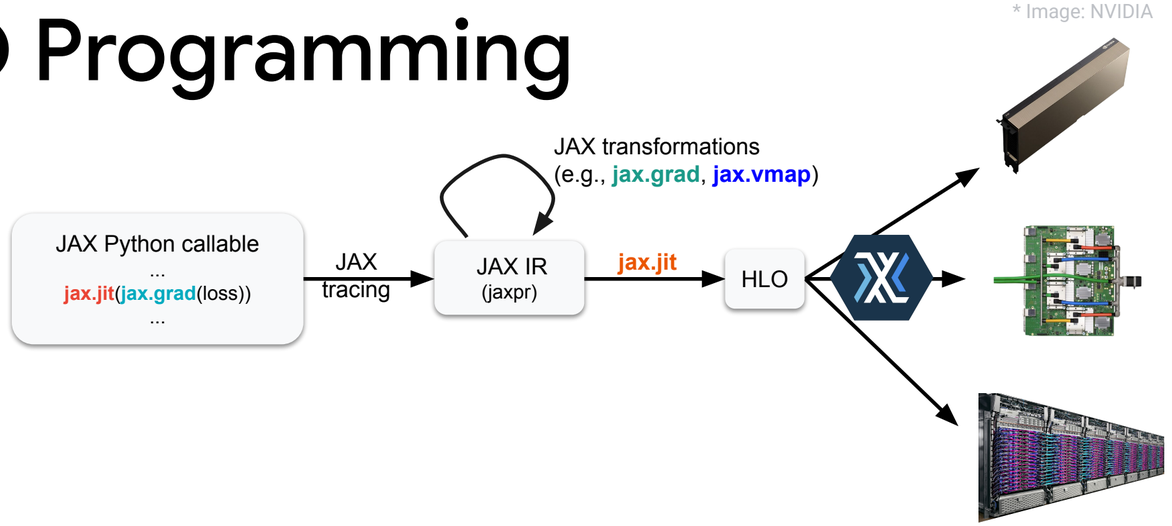


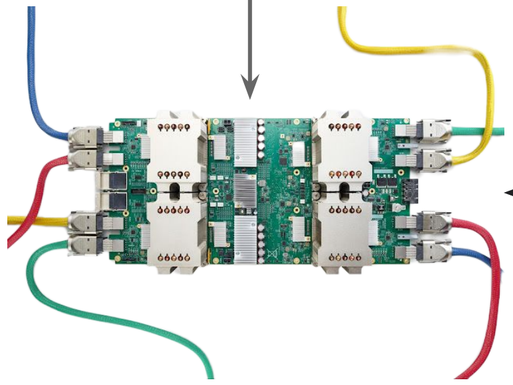
Pallas
JAX kernel language

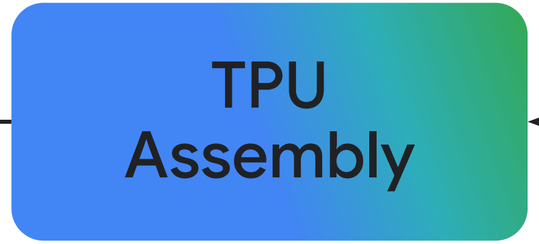
```
def add_kernel(x_ref, y_ref, out_ref):
  out_ref[:, :] = x_ref[...] + y_ref[...]
```

Pallas: An Alternative to HLO Programming

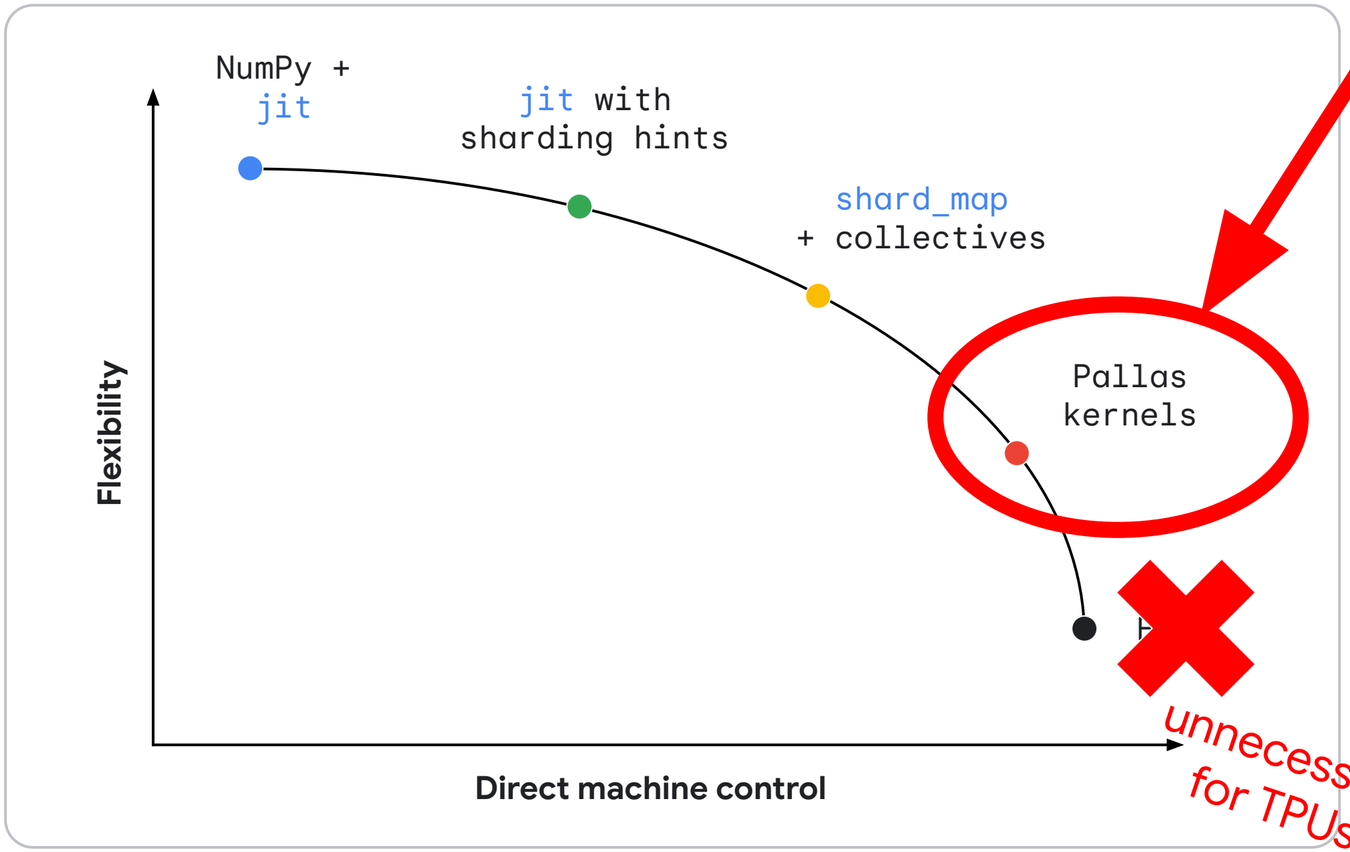
* Image: NVIDIA







Pallas: Low-level control



Pallas

Math

- Compiler, take the wheel!
- Here's a hint
- I'll handle comms
- **Kernel languages**
- FFI

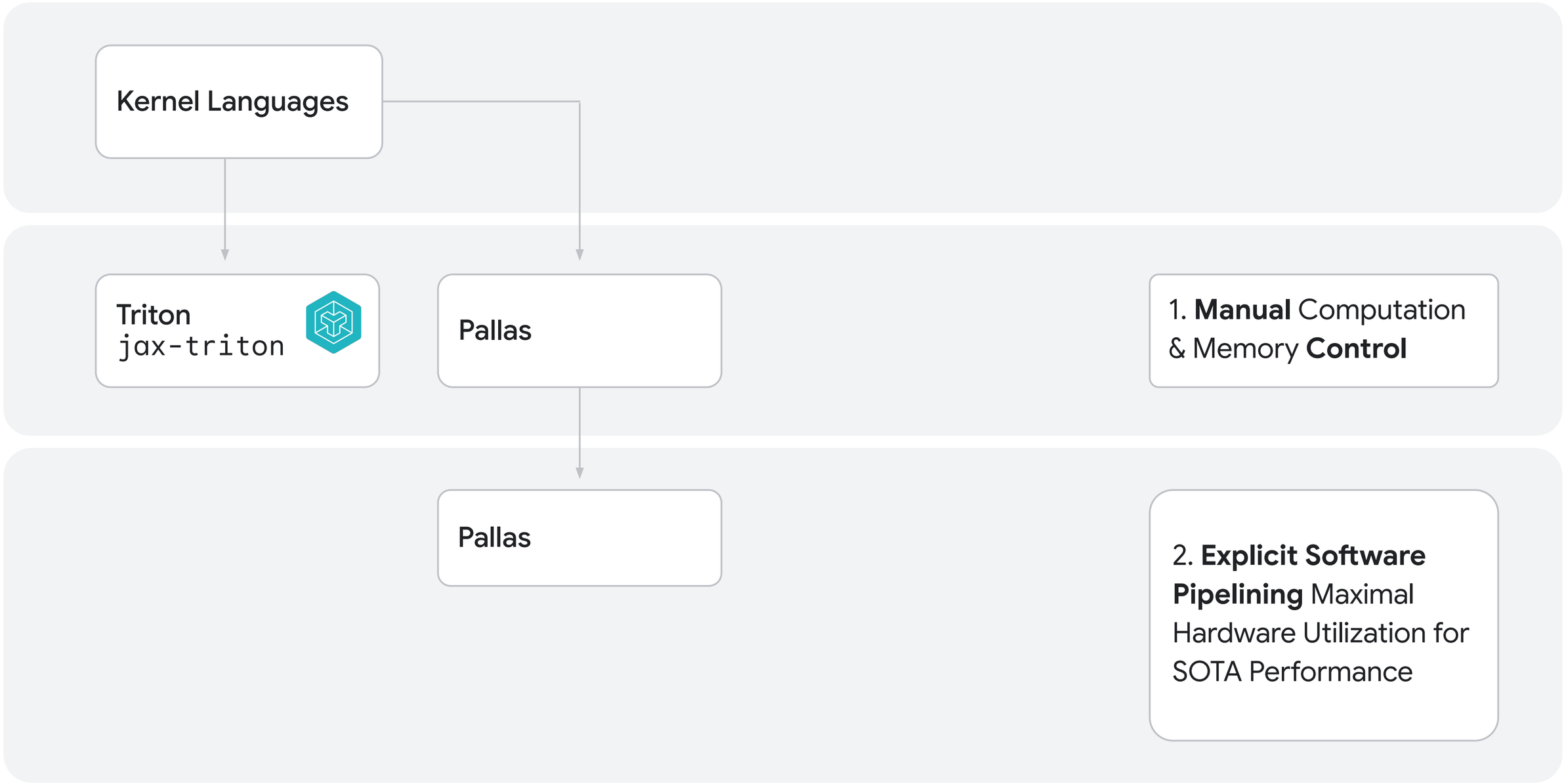
Physics

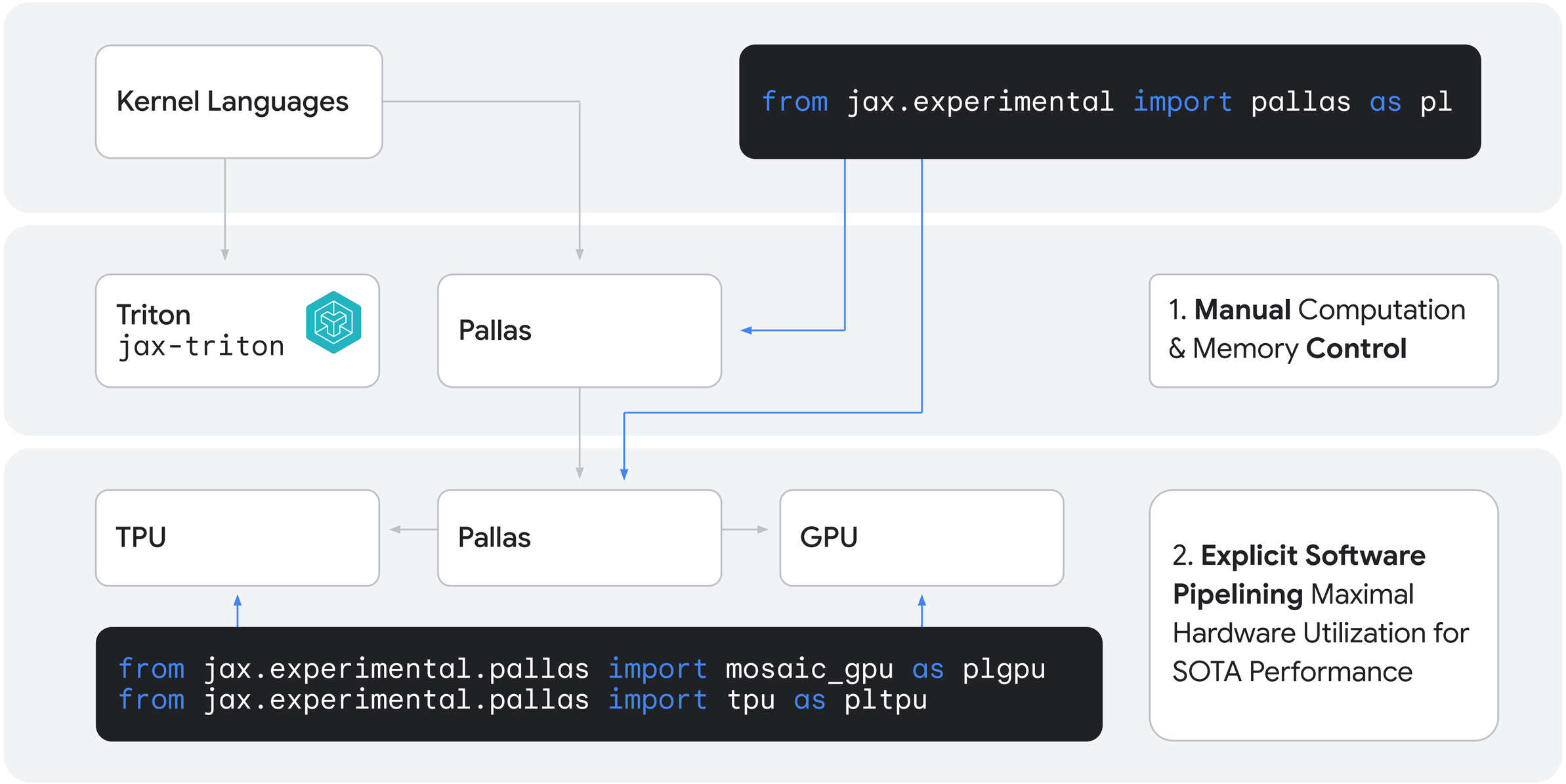
Why Pallas?

- want more control?
  - direct hardware control
  - exploit structured sparsity in your problem
- explicit control over memory and compute usage

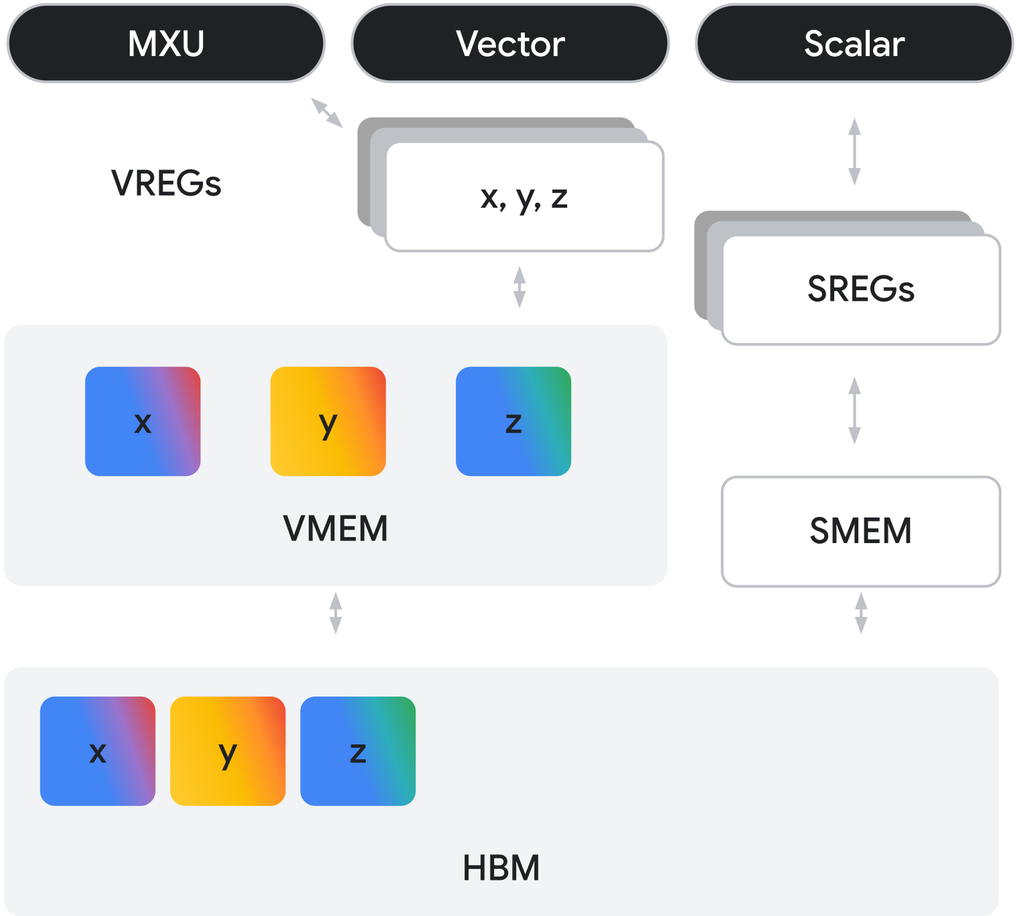
Example applications:

- Block-sparse linear algebra
- Mixture-of-Experts
- Custom communication
- Low-level loops


Kernel Languages
Triton
jax-triton
Pallas
Pallas
1. **Manual** Computation & Memory **Control**
2. **Explicit Software Pipelining** Maximal Hardware Utilization for SOTA Performance


Kernel Languages
from jax.experimental import pallas as pl
Triton
jax-triton
Pallas
1. Manual Computation & Memory Control
TPU
Pallas
GPU
2. Explicit Software Pipelining Maximal Hardware Utilization for SOTA Performance
from jax.experimental.pallas import mosaic_gpu as plgpu
from jax.experimental.pallas import tpu as pltpu

Pallas
Memory pipeline

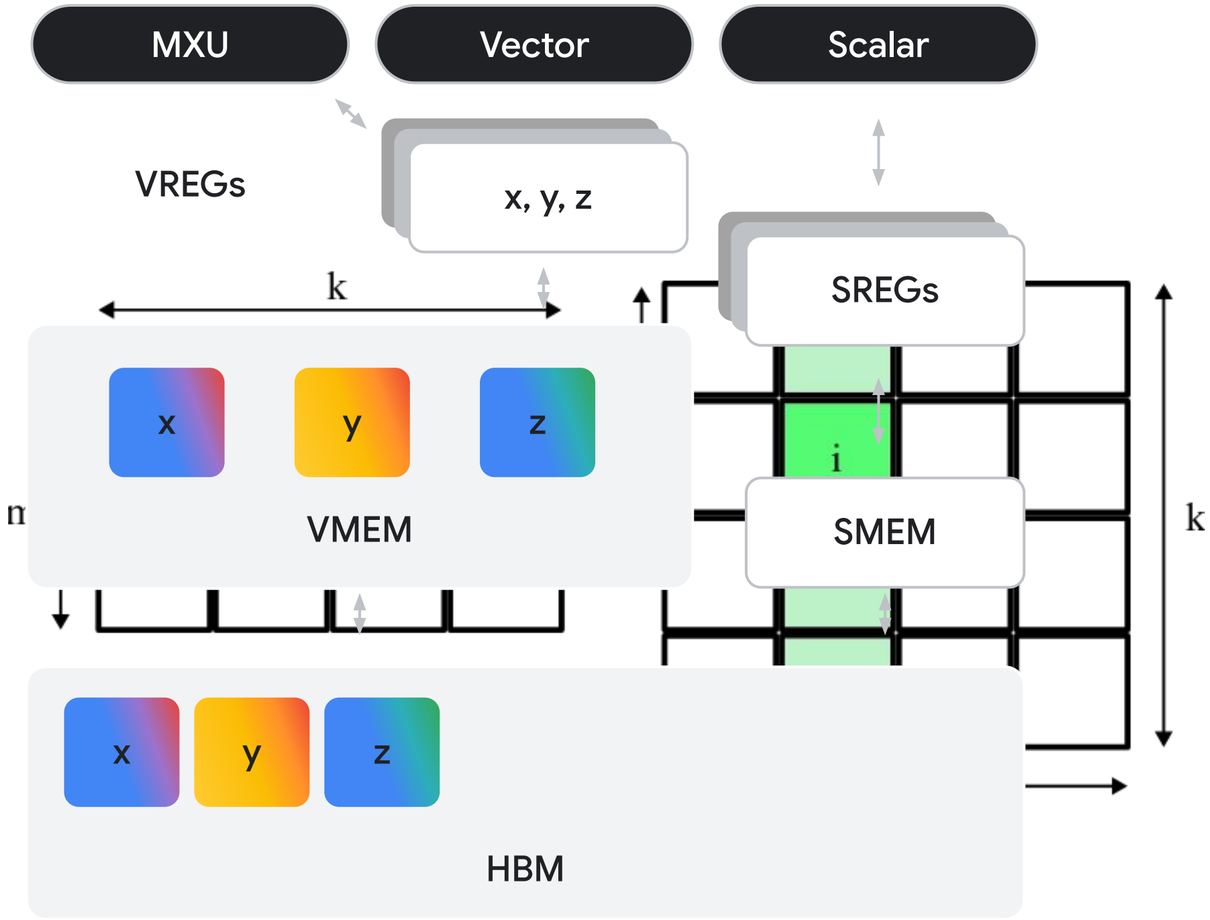



```
def add_kernel(x_ref, y_ref, out_ref):
  x = x_ref[:, :]
  y = y_ref[:, :]
  out = x + y
  out_ref[:, :] = out
```

Pipeline HBM ↔ VMEM

```
jax_add_kernel = pl.pallas_call(
  add_kernel,
  out_shape=jax.ShapeDtypeStruct(x.shape, x.dtype),
  in_specs=[pl.BlockSpec((m, n), lambda i, j: (i, j)),
            pl.BlockSpec((m, n), lambda i, j: (i, j))],
  out_specs=pl.BlockSpec((m, n), lambda i, j: (i, j)),
  grid=(M, N),
)
```

Pallas
Matmuls are very simple

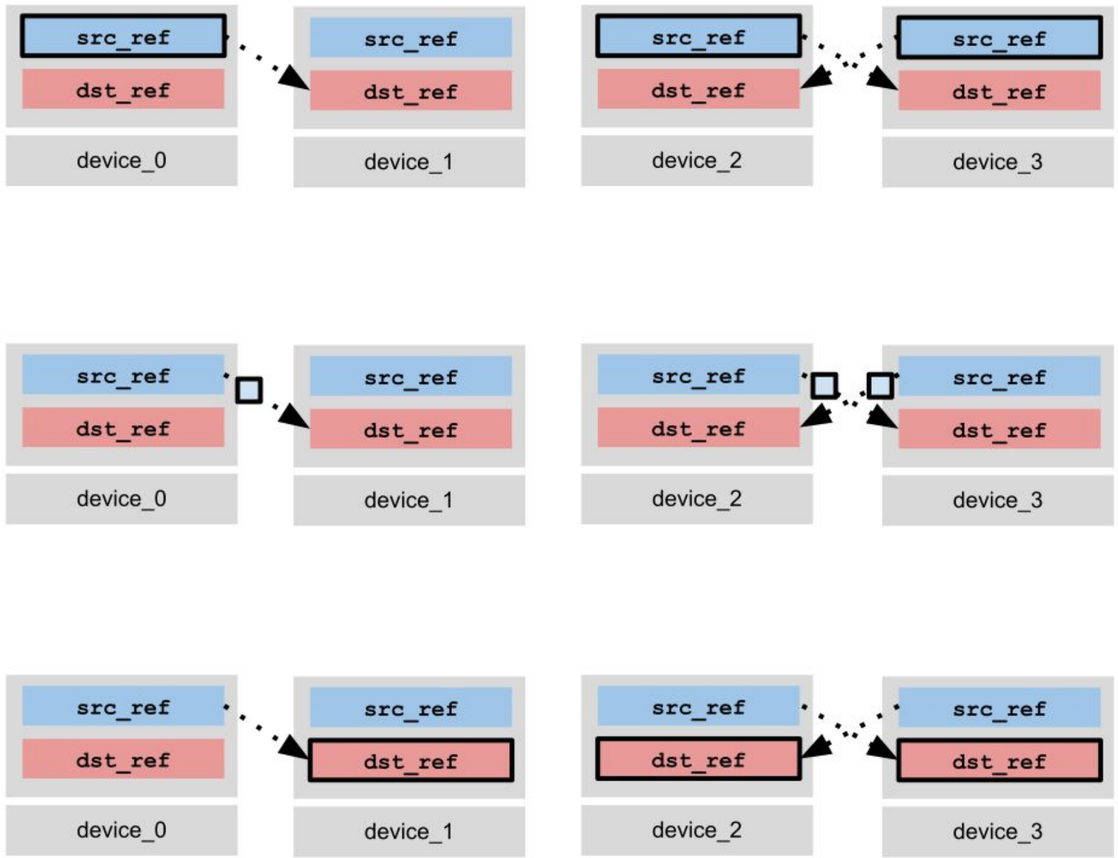



```
def matmul_kernel(x_ref, y_ref, out_ref):
  out_ref[...] = pl.dot(x_ref[...], y_ref[...])
```

Pipeline HBM ↔ VMEM

```
def matmul(A, B, bm, bn, bk):
  out_shape = jax.ShapeDtypeStruct((m, n), "bfloat16"),
  in_specs = [
      pl.BlockSpec((bm, bk), lambda i, j, k: (i, k)),
      pl.BlockSpec((bk, bn), lambda i, j, k: (k, j))
  ]
  out_specs = pl.BlockSpec(
      (bm, bn), lambda i, j, k: (i, j)
  )
  grid = (
    pl.cdiv(m, bm), pl.cdiv(n, bn), pl.cdiv(k, bk)
  )
  return pl.pallas_call(
      matmul_kernel,
      out_shape=out_shape,
      grid=grid,
      in_specs=in_specs,
      out_specs=out_specs,
  )(A, B)
```

Pallas
Inter-chip Communication
RDMAs




```python
def example_kernel(input_ref, output_ref, send_sem, recv_sem):
    device_id = lax.axis_index('x')
    copy_0_to_1 = pltpu.make_async_remote_copy(
        src_ref=input_ref,
        dst_ref=output_ref,
        send_sem=send_sem,
        recv_sem=recv_sem,
        device_id=1,
    )
    copy_2_to_3 = pltpu.make_async_remote_copy(
        src_ref=input_ref,
        dst_ref=output_ref,
        send_sem=send_sem,
        recv_sem=recv_sem,
        device_id=3,
    )
    copy_3_to_2 = pltpu.make_async_remote_copy(
        src_ref=input_ref,
        dst_ref=output_ref,
        send_sem=send_sem,
        recv_sem=recv_sem,
        device_id=2,
    )
    @pl.when(device_id == 0)
    def _():
      copy_0_to_1.start()
      copy_0_to_1.wait_send()
    @pl.when(device_id == 1)
    def _():
      copy_0_to_1.wait_recv()
    @pl.when(device_id == 2)
    def _():
      copy_2_to_3.start()
      copy_2_to_3.wait_send()
      copy_3_to_2.wait_recv()
    @pl.when(device_id == 3)
    def _():
      copy_3_to_2.start()
      copy_3_to_2.wait_send()
      copy_2_to_3.wait_recv()
```

docs.jax.dev/en/latest/pallas/tpu/distributed.html

Example: all-gather

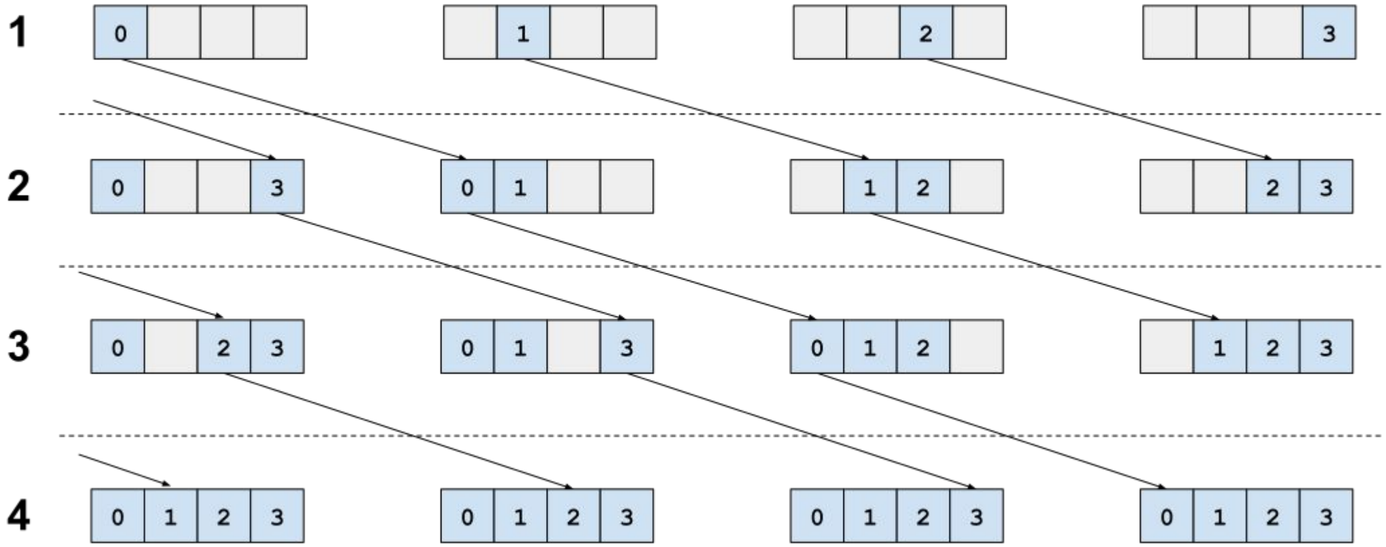
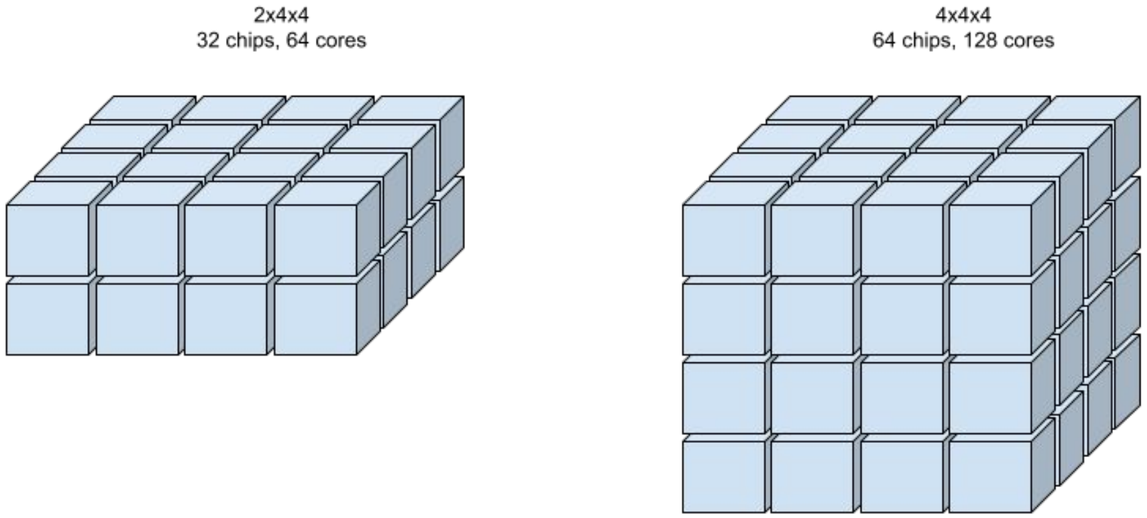


Fig: All-gather via permute by remote direct memory access (RDMA)



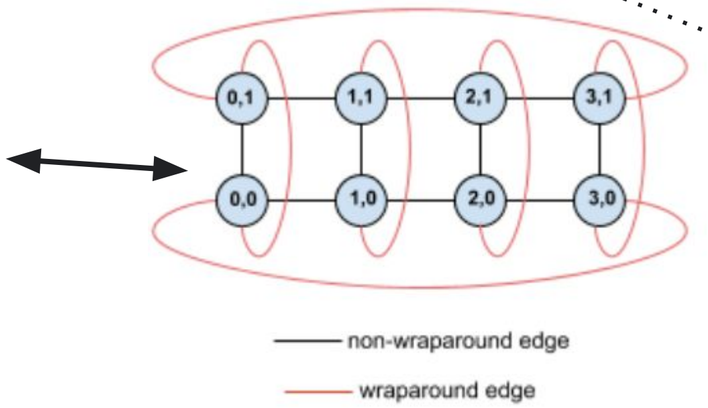
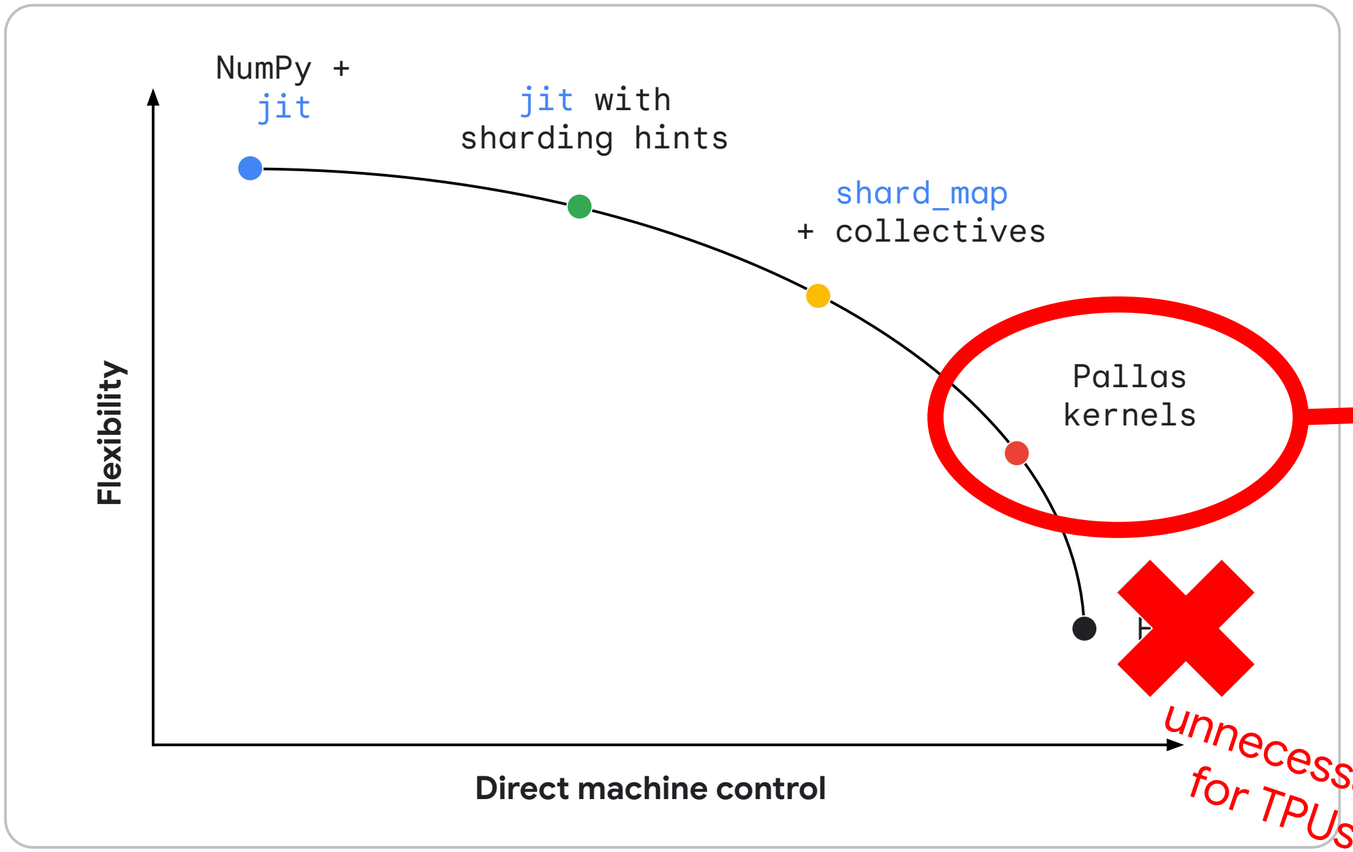
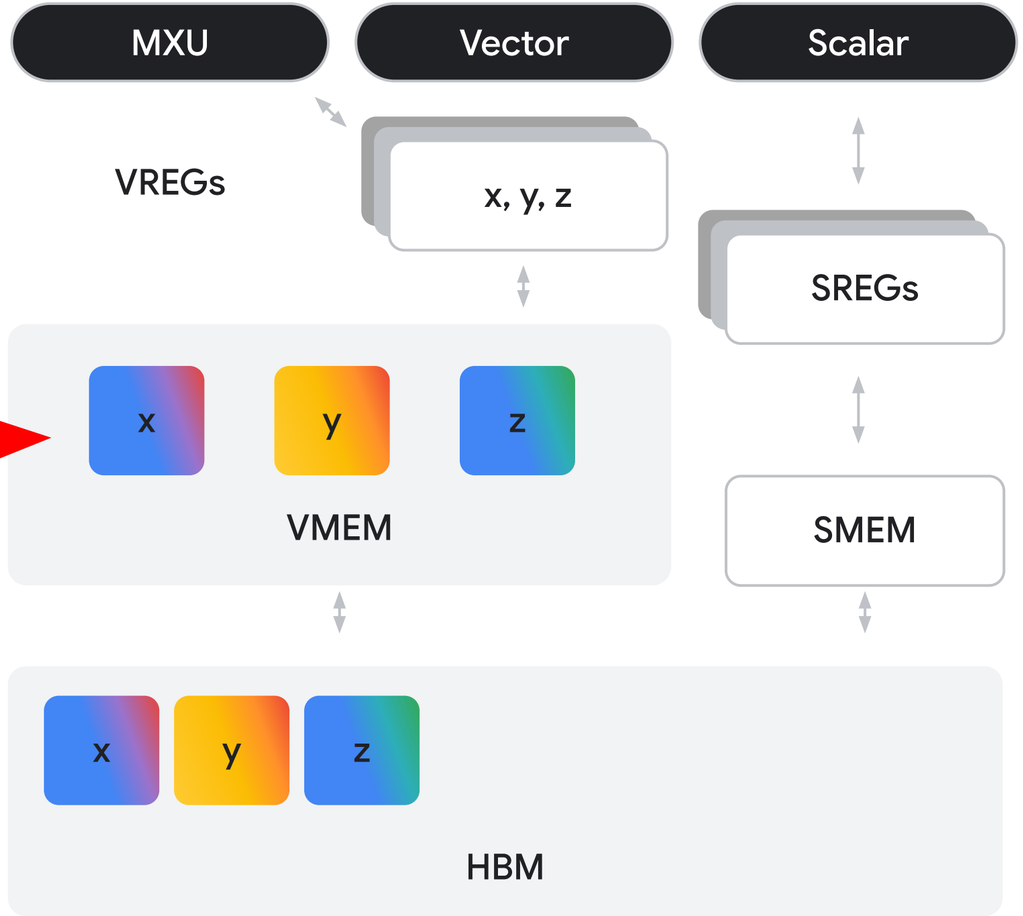


Fig: Example TPU topologies

Summary: JAX + Pallas + TPUs



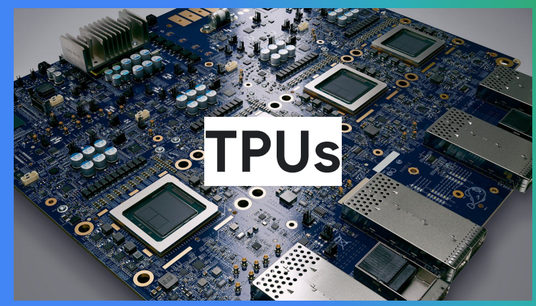


Resources

- JAX Docs & Guides: jax.dev
- JAX AI stack: jaxstack.ai
- A Guide on Scaling: jax-ml.github.io/scaling-book/

Pallas
JAX kernel language

```
def add_kernel(x_ref, y_ref, out_ref):
  out_ref[:, :] = x_ref[...] + y_ref[...]
```



Thank you!

Q&A

Please ask questions on the RSVP site