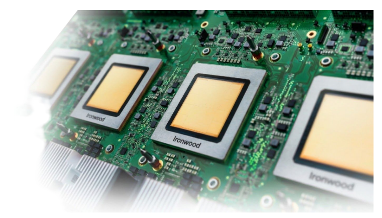


Running PyTorch natively on Google TPUs

PyTorch Conference Europe 2026

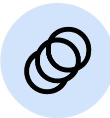
Jana van Greunen
Director of PyTorch Engineering
janavg@meta.com

Claudio Basile
Tech Lead for TorchTPU
cbasile@google.com

Kat Ko
Eng Lead for TorchTPU
caffleine@google.com

PyTorch and HW Optionality

The AI Infrastructure Challenge Has Evolved

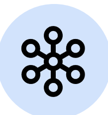


Heterogeneous Hardware Needs

Training, post-training, inference – demand different compute profiles.

GPUs, TPUs, and custom chips each bring unique strengths.

Mixed fleets are the new standard.



From Single GPU to 100K+ Clusters

Frontier models require distributed systems spanning thousands of accelerators with fault tolerance built in.

At scale, **hardware portability** is an operational necessity.

PyTorch: The Portable Foundation

As the #1 ML framework, PyTorch is the natural unifying layer.

Researchers and developers should be able to **write their model once** and **run it on the best hardware** for the job.

PyTorch on TPU Stack

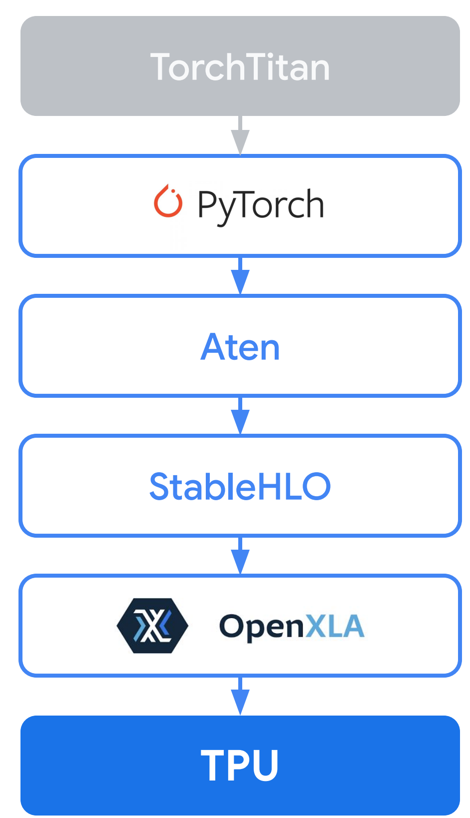
Leading with Usability, Portability, Performance

PyTorch-Native

Migrate your existing workloads with minimal code change. Deeply integrated into the PyTorch ecosystem.

Built for Scale

Architected to handle massive workloads, supporting thousands of TPUs, and next generation distributed methods.



Google

Usability: Getting started on TPU

```
% pip install torch

import torch

…
my_tensor.to("cuda")
my_model.to("cuda")
…
```

Today Standard:
Existing model, accelerated on GPU

```
% pip install torch

import torch

…
my_tensor.to("tpu")
my_model.to("tpu")
…
```

TorchTPU Tomorrow:
Workload ported to TPU: Future UX goal, depends on the upcoming PyTorch support for Wheel Next

TorchTPU is built from the ground up to ease migration to TPU

Google

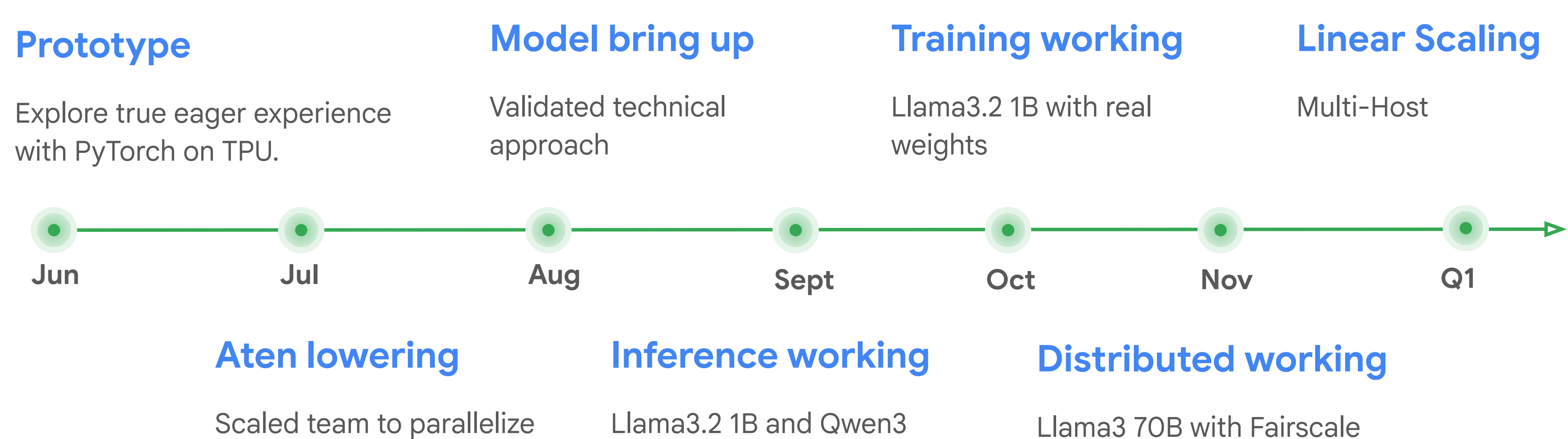
Usability: More advanced use cases

		PyTorch	**TorchTPU**	**JAX, TorchXLA, TorchAX**
Distributed	Set up	**One process per device**	**One process per device**	One process per host
	Workload types	**MPMD + SPMD**	**MPMD + SPMD**	SPMD only
	Sharding	**DTensor, FSDP**, …	**DTensor, FSDP**, …	gSMPD
	Orchestration	**Ray, Monarch**, …	**Ray, Monarch**, …	Pathways
Kernels	DSL	CUDA/cuTile, Triton, **Helion**	Pallas, **Helion**	Pallas

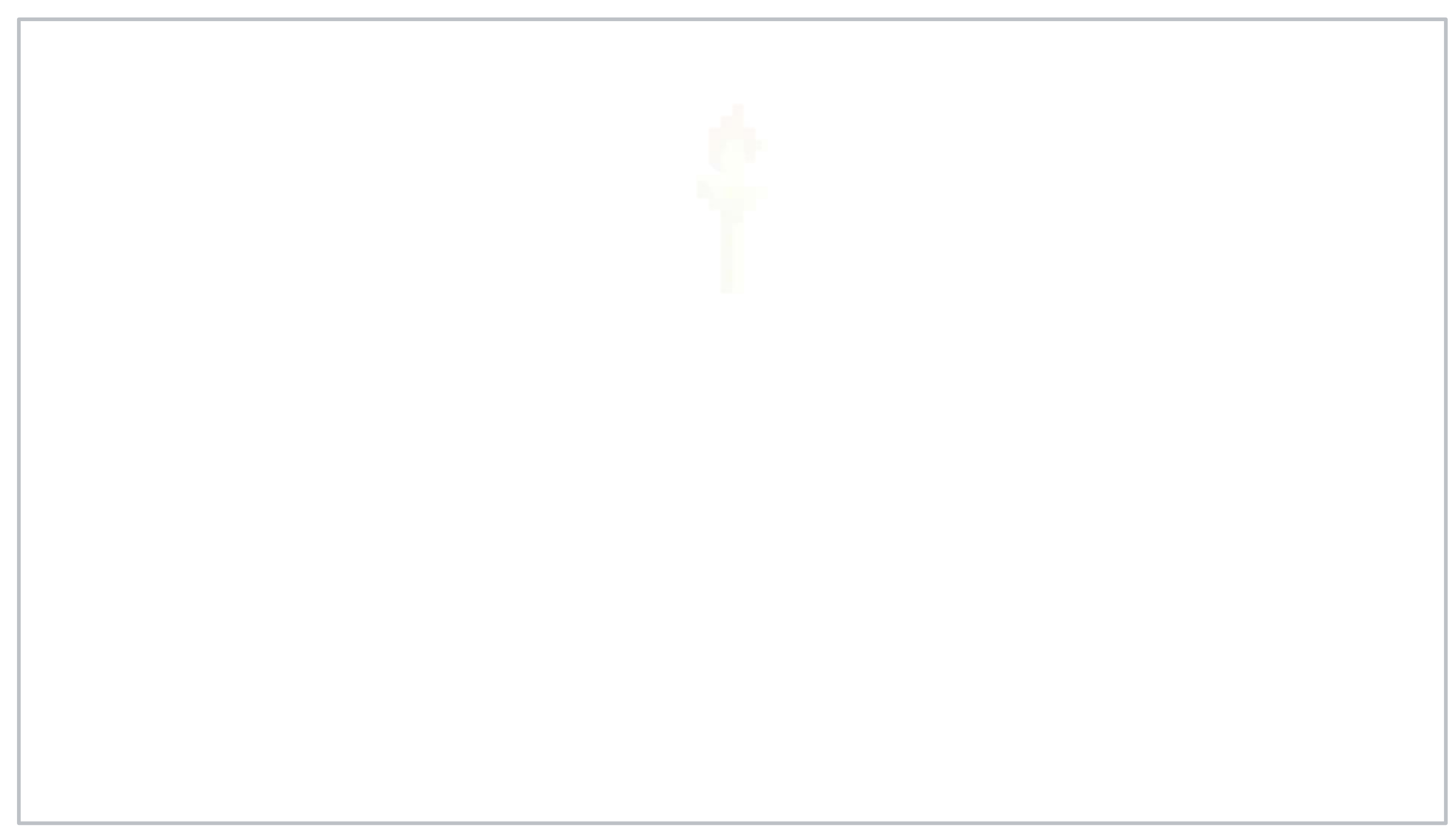
TorchTPU is built from the ground up to support PyTorch idioms

Google

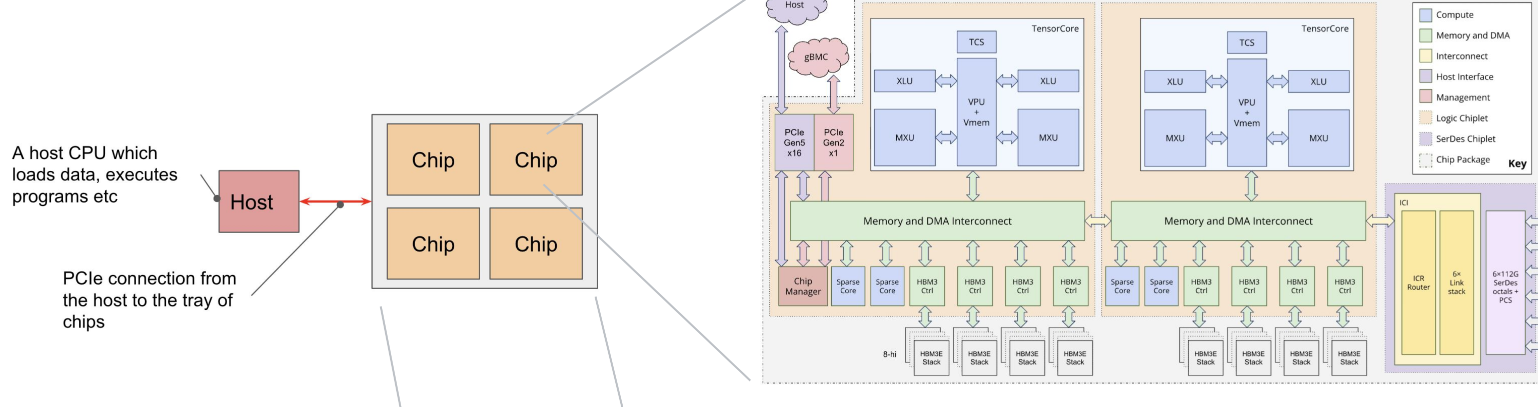
Moving from concept to scaled Native Pytorch on TPU



TorchTPU Demo

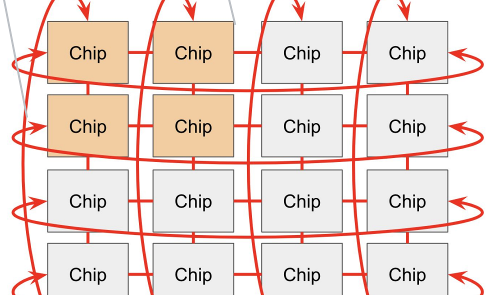


Google



Topology:

- A host is attached to multiple chips
- A chip is connected to host via **PCIe** and to other chips via **ICI**
- ICI connects chips in a 2D/3D Torus structure



A chip includes X **TensorCores** and Y **SparseCores**

- TensorCore is for dense math and is **single-threaded**
- SparseCore is for irregular memory access and computation (e.g., embeddings, gather/scatter) and for offloading ops from TC (e.g., collectives)

More info [here]

Google

Eager First, all else will follow

- Implemented as a PrivateUse1 backend

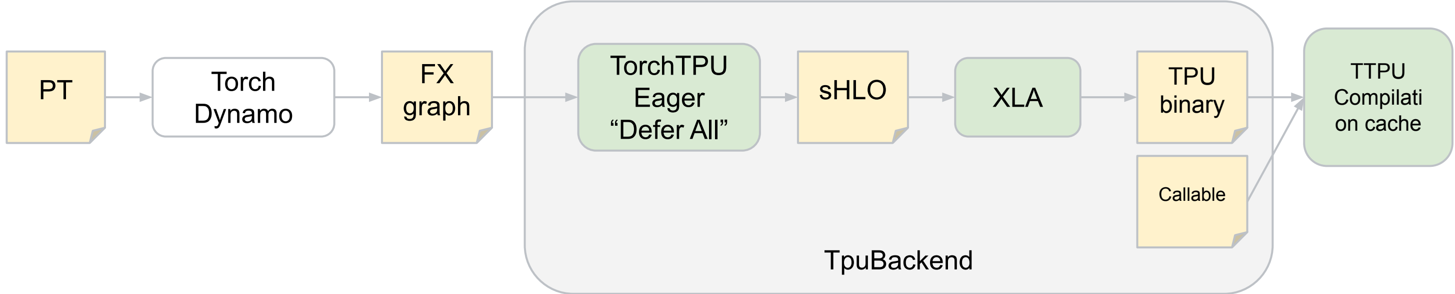
- Multiple eager modes
  - **Debug**: Dispatch one op at the time, but sync execution
    - Slow, similar to CUDA_LAUNCH_BLOCKING, plus bounds checks
  - **Strict**: Dispatch one op at the time + async execution
    - Similar to default PyTorch on GPU
  - **DeferAndFuse**: Defer and fuse multiple ops into larger chunks, through automated heuristics
    - New eager modality with TorchTPU, 1.5-2x faster than Strict Eager

- Shared **Compilation Cache**
  - Single host, across workers
  - Optionally persistent and/or multi-host

Google

Torch.Compile

We start with Dynamo, but use XLA vs Inductor



Benefits vs using Inductor:

- Faster development path for us (it reuses TorchTPU eager path)
- XLA is battle tested for TPU
- XLA can optimize overlap between collectives and computation

Kernels

The **Pallas** kernel ✅

```
from torch_tpu._internal import pallas

@pallas.custom_kernel(lambda x, y: torch.empty_like(x))
def add_vectors(x_ref, y_ref, o_ref):
  x, y = x_ref[...], y_ref[...]
  o_ref[...] = x + y
```

The **JAX** kernel ✅

```
from torch_tpu._internal import pallas

@pallas.custom_jax_kernel
  def add_vectors(x, y):
   return jax.numpy.add(x, y)
```

The **Helion** kernel (coming up...)

```
@helion.kernel
def add_vectors(x: torch.Tensor, y: torch.Tensor) -> torch.Tensor:
    out = torch.empty_like(x)
    for tile in hl.tile(x.shape[0]):
        out[tile] = x[tile] + y[tile]
    return out
```

The user experience

```
def test_kernel(self):
    x = torch.tensor([0.1, 0.2, 0.3], device=tpu_d)
    y = torch.tensor([0.4, 0.5, 0.6], device=tpu_d)
    expected = torch.add(x, y).to("cpu")
    actual = add_vectors(x, y).to("cpu")
    utils.assert_close(actual, expected)
```

Distributed

- ✅ Most of ProcessGroup, Subgroups API, and major collectives implemented.
- ✅ DDP (distributed-data parallel) - manual and wrapper-based.
- ✅ PyTorch’s DTensor *just works*!
- ✅ Tensor-parallel via *3rd-party* Fairscale library (zero patches!)
- ✅ Llama3: 8B / 70B distributed inference (Meta’s ref impl with TP)
- ✅ Various HF model implementations (Llama3, gpt-oss-120b, Qwen3)
- ✅ FSDPv2
- ✅ Distributed training on multi-host
- ✅ MPMD + SPMD
- ⌛ Performance / scalability
- ⌛ Streams and events

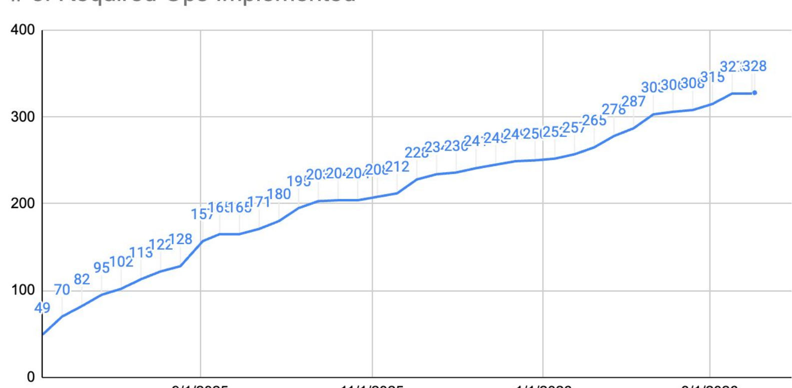
Results: *A downloaded model should just work…*

Automatic coverage testing:

- Downloaded **4,315** models from Hugging Face
- **2,968** (**68.8%)** models ran on CPU and TPU
- **19** models had bugs (**5** unique)
- **289** Models use unsupported ops (**14** missing)
- Remainder are script, system, model issues

Here are some examples:

- CLIPModel, CLIPVisionModel
- DeepseekV3ForCausalLM
- Gemma3ForCausalLM
- GPT OSS 10B
- Qwen2ForCausalLM, Qwen3ForCausalLM, Qwen3 0.6B, Qwen3MoeForCausalLM
- Llama-3.2-1B Fwd / Bwd, Llama4forCausalLM
- SDXL DiffusionPipeline
- Resnet40



We are also tracking/improving performance on several showcase models:

- Llama3, Qwen3, Gemma3, …, various configs

Results: *Performance on Hugging Face models*

Out of the box TorchTPU performance in various configurations

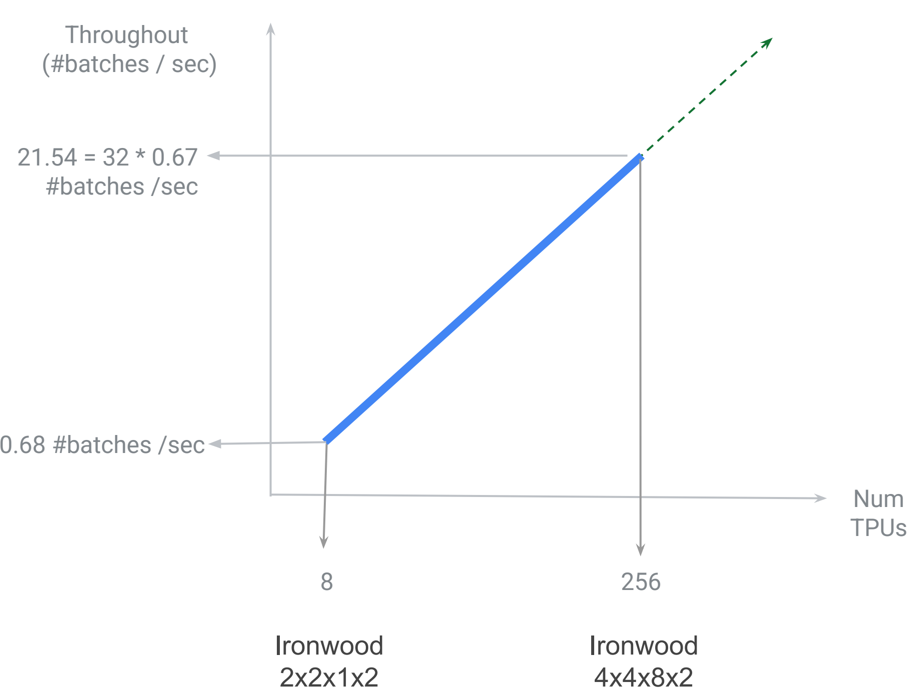
Model	**Debug** (one op at the time + sync + bounds checks)	**Strict** (one op at the time + async)	**DeferAndFuse** (fused op clusters + async)	**Compiled** (dynamo fx graph)
Gemma3 270m 1 TPU Train step	2.9s	0.64s	0.29s	0.047s
Llama3.2 1B 1 TPU Fw step	0.31s	0.19s	0.10s	0.067s
Llama3.2 1B 1 TPU Train step	3.4s	2.0s	1.4s	0.85s
Llama3.2 1B 8 TPUs with FSDP Train step	48s	40s	25s	25s

Results: *Performance scales linearly with # of TPUs*

Basic training loop with:

- Hugging Face Llama3.2 1B
- Bfloat16
- FSDPv2
- Real weights
- Random data
- 2056 sequence length
- 10 train_steps
- 16 batch size

Similar results obtained for Llama3.2 8B



TPU Hardware Awareness

ML models may be affected by an intrinsic bias they will be run on GPU:

- Head dimension = 64 (vs 128 or 256)
- Use of data-dependent ops such as torch.masked_select, torch.nonzero (e.g., for MoE)
- Unnecessary / excessive use of view ops
- Channel first vs channel last

Portable software does not mean the realities of the hardware disappear, if peak performance are expected

We start with portability and correctness, then provide:

- **Tools** to identify perf bottlenecks (xprof) and **APIs** to apply optimizations (e.g., custom kernels)
- **Performance guidelines** to ease the TPU transition, specifically for PyTorch users (in the meantime, go ahead and read https://jax-ml.github.io/scaling-book/)

Open Challenges

- Avoiding recompilations due to shape changes ⇒ XLA bounded dynamism
- Avoiding longer first iteration ⇒ Library of precompiled TPU kernels for Aten ops
- Validating linear scaling to Pod-size and beyond
- What's the Pathways / Shardy equivalent for PyTorch?

What’s coming up in 2026

In no specific order:

- Public Github repository with documentation, tutorials, and examples
- Helion
- vLLM
- TorchTitan
- Dynamic shapes support with torch.compile
- A “tpuBLAS” library
- Scaling to Pod-size and beyond
- Performance optimization for additional model architectures (Diffusion, Recommenders, …)
- CUDA stream emulation

Sign Up for our Newsletter to get the latest on TorchTPU

APPENDIX

PyTorch + Google Principles

Usability

It should feel native to PyTorch users and fit in the PyTorch ecosystem.

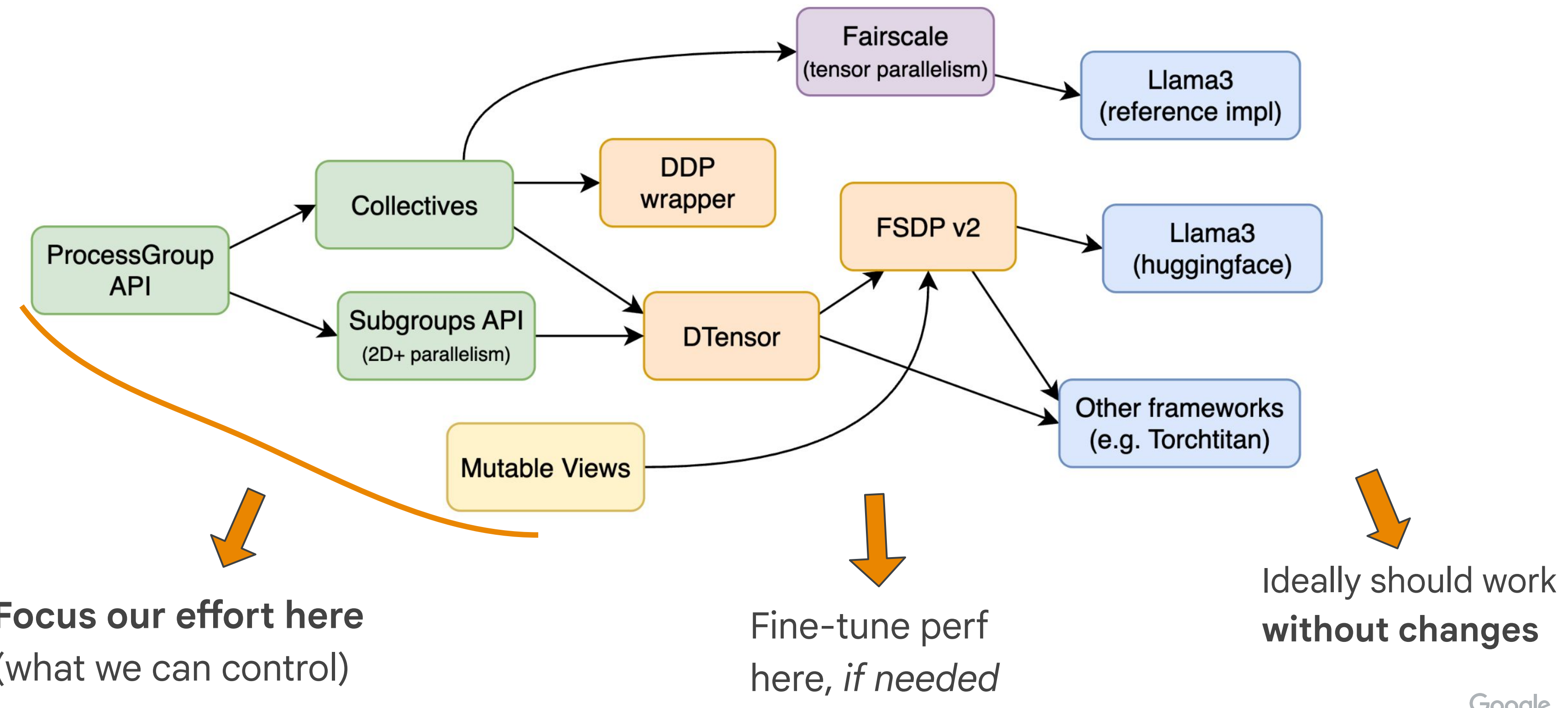
Portability

It should run PyTorch workloads out of the box, with minimal/trivial changes.

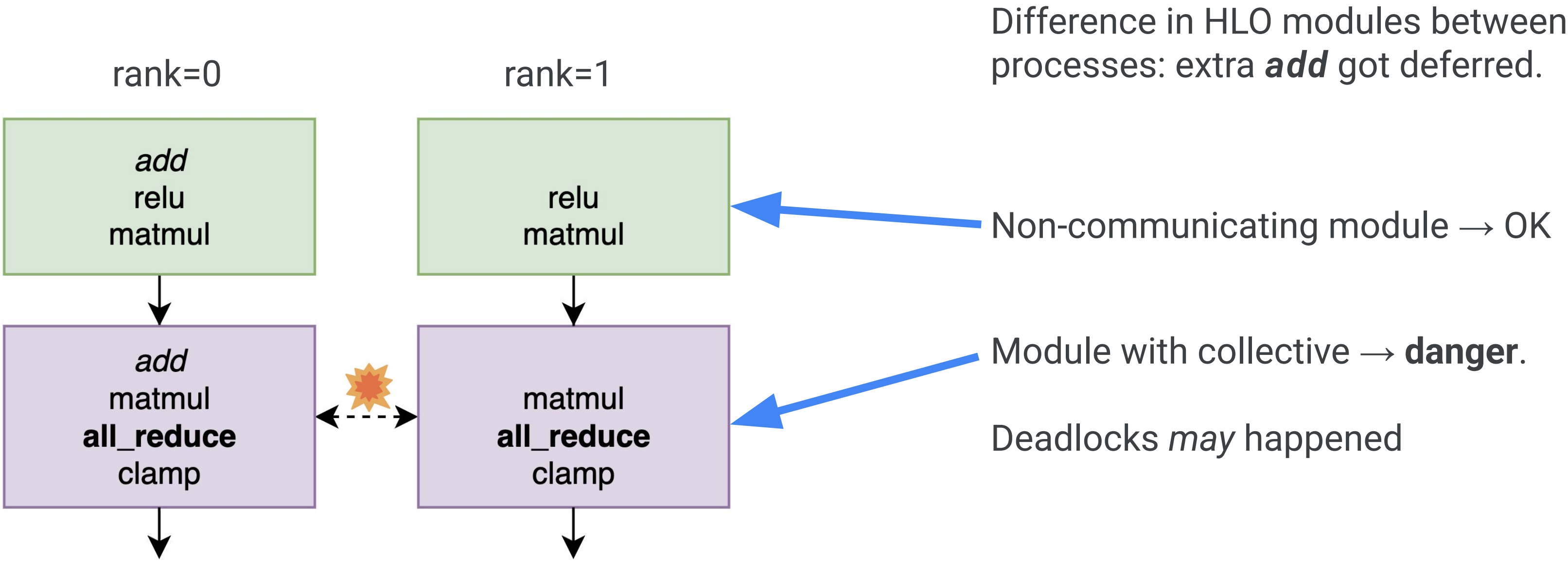
Performance

It should allow PyTorch users to extract peak TPU performance.

Distributed



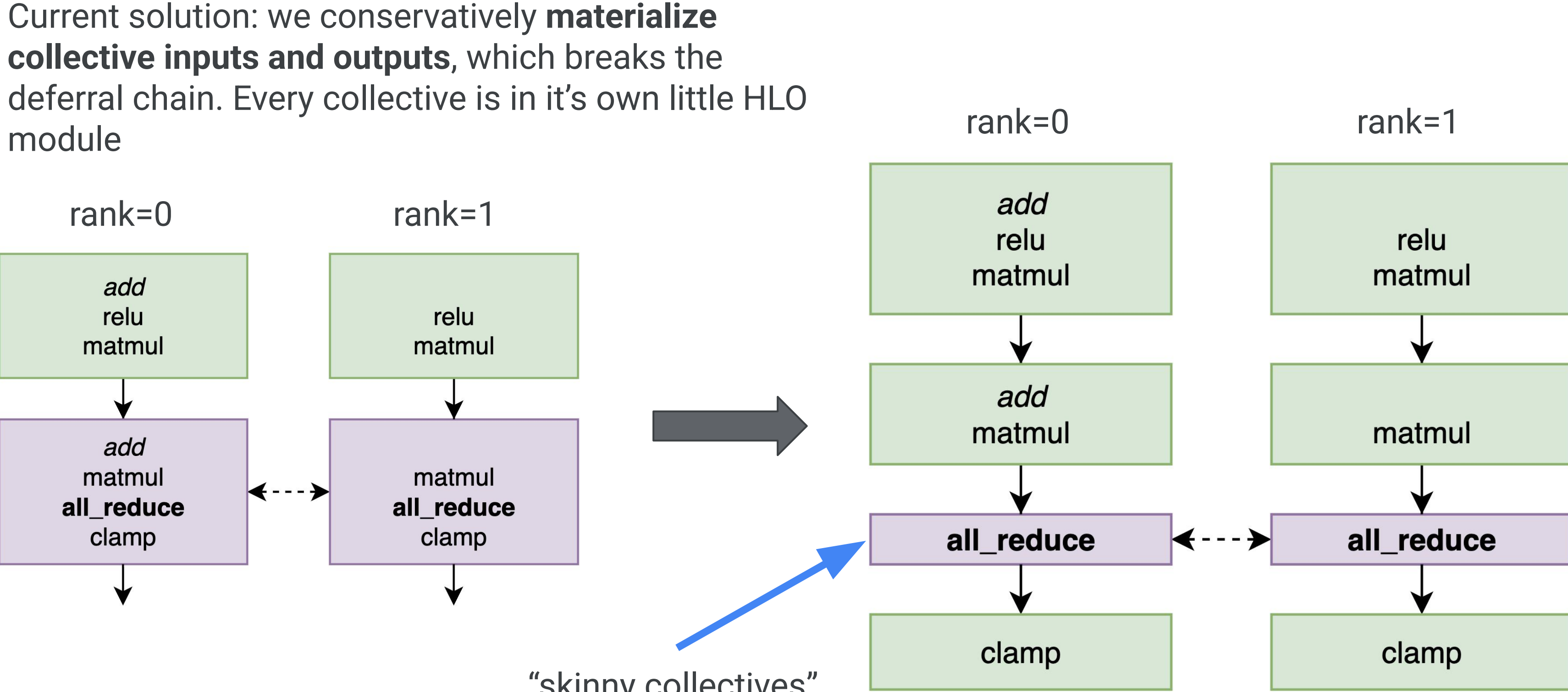
MPMD: “Safety” materialization around collectives



Google

MPMD: "Safety" materialization around collectives

Current solution: we conservatively **materialize collective inputs and outputs**, which breaks the deferral chain. Every collective is in it's own little HLO module



Google

PyTorch with CUDA backend

```python
import torch
import torch.distributed as dist
from torch.distributed.fsdp import FullyShardedDataParallel as FSDP
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset, DistributedSampler
from transformers import AutoModelForCausalLM

def setup_dist():
 dist.init_process_group(backend="nccl")

def simple_example():  # pylint: disable=missing-function-docstring
 # Distributed device setup
 setup_dist()
 rank = dist.get_rank()
 device = torch.device("cuda", rank)
 torch.cuda.set_device(device)

 model = AutoModelForCausalLM.from_pretrained("meta-llama/Meta-Llama-3-8B")
 model = FSDP(model, device_id=rank, auto_wrap_policy=...)
 optimizer = optim.AdamW(model.parameters(), lr=1e-4)

 # Dataset and DataLoader
 dataset = ...
 sampler = DistributedSampler(dataset, ...)
 dataloader = DataLoader(dataset, ...)

 # Training loop
 num_epochs = ...
 model.train()
 for epoch in range(num_epochs):
   sampler.set_epoch(epoch)
   for batch in dataloader:  # pylint: disable=unused-variable
     # Move batch to device
     input_ids, labels = ...  # pylint: disable=unpacking-non-sequence
     optimizer.zero_grad()
     outputs = model(input_ids=input_ids, labels=labels)
     loss = outputs.loss
     loss.backward()
     optimizer.step()
```

Changes for PyTorch/XLA

```python
import torch
import torch.distributed as dist
from torch.distributed.fsdp import FullyShardedDataParallel as FSDP
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset, DistributedSampler
from transformers import AutoModelForCausalLM

def setup_dist():
 dist.init_process_group("xla", init_method='xla')

def simple_example():  # pylint: disable=missing-function-docstring
 # Distributed device setup
 setup_dist()
 rank = dist.get_rank()
 device = torch.device("xla", rank)
 torch.cuda.set_device(device)

 model = AutoModelForCausalLM.from_pretrained("meta-llama/Meta-Llama-3-8B")
 model = FSDP(model, device_id=rank, auto_wrap_policy=...)
 optimizer = optim.AdamW(model.parameters(), lr=1e-4)

 # Dataset and DataLoader
 dataset = ...
 sampler = DistributedSampler(dataset, ...)
 dataloader = DataLoader(dataset, ...)

 # Training loop
 num_epochs = ...
 model.train()
 for epoch in range(num_epochs):
   sampler.set_epoch(epoch)
   for batch in dataloader:  # pylint: disable=unused-variable
     # Move batch to device
     input_ids, labels = ...  # pylint: disable=unpacking-non-sequence
     optimizer.zero_grad()
     outputs = model(input_ids=input_ids, labels=labels)
     loss = outputs.loss
     loss.backward()
     optimizer.step()
     torch_xla.mark_step()
```

Why TPUs?

Mature Stack

Production-hardened stack tested across O(10k) models.

Diverse Support

Running inference & training workloads across Google Cloud customers.

Hero Scale

Best hero example: Gemini, showcasing state-of-the-art capability.

TPU ✦ Background

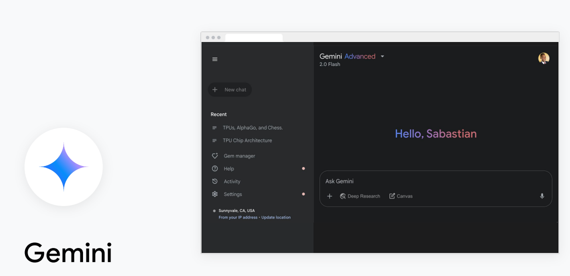
Custom ASICs are the foundation of the supercomputing infrastructure designed by Google, specifically for ML and AI

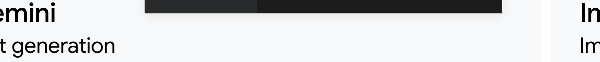
Deliver best in class perf/$ and perf/W for key AI workloads

Enable Largest, Most-Reliable, and Fastest-to-Ramp AI Infra

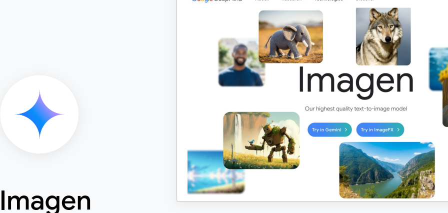
Drive seamless customer experience from evaluation to production

How Google uses TPU





Gemini
Text generation



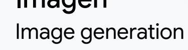
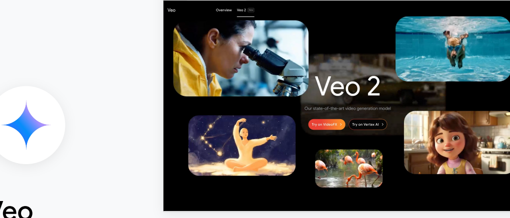
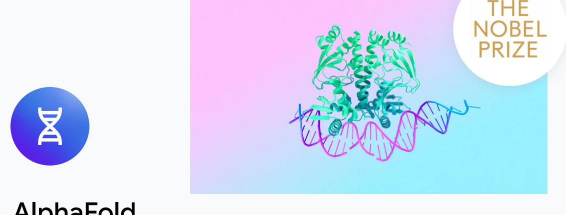


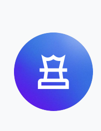
Imagen
Image generation

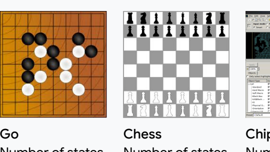


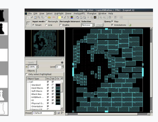
Veo
Video generation



AlphaFold
Protein 3D structure prediction

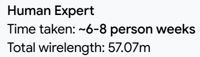






Go	Chess	Chip placement
Number of states	Number of states	Number of states
$\sim 10^{123}$	$\sim 10^{360}$	$\sim 10^{9000}$

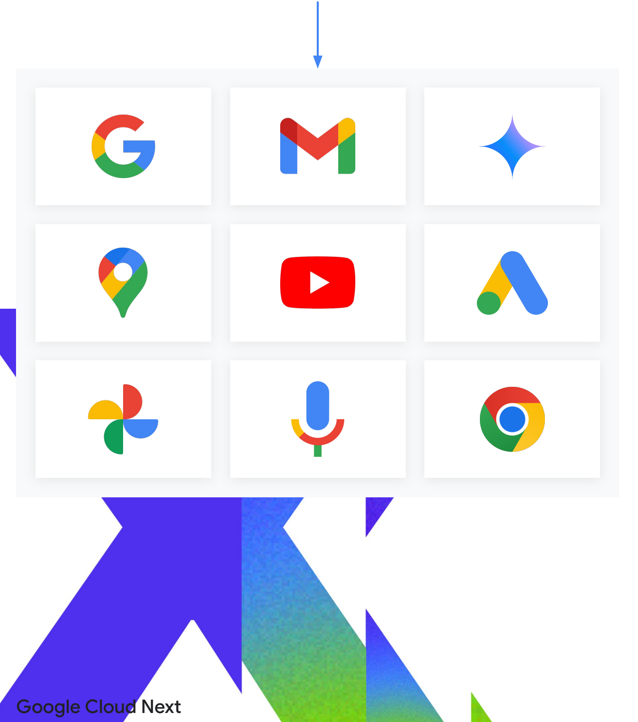
AlphaGo/Zero
Human-rivaling gameplay

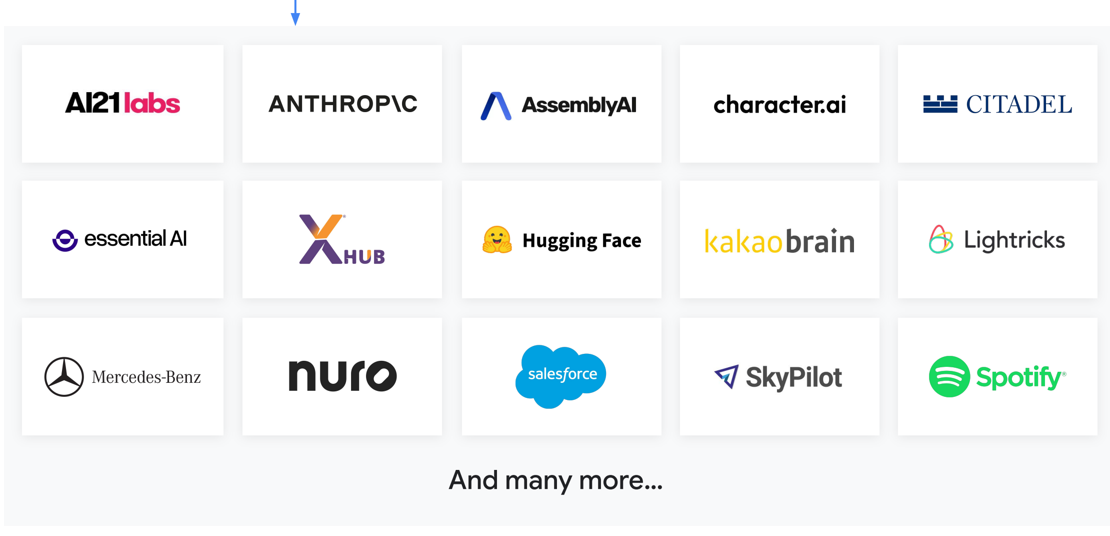


Human Expert	ML Placer
Time taken: **~6-8 person weeks**	Time taken: **24 hours**
Total wirelength: 57.07m	Total wirelength: 55.42m

AlphaChip
Chip-designing agent

TPU powers AI for Google and our Cloud customers

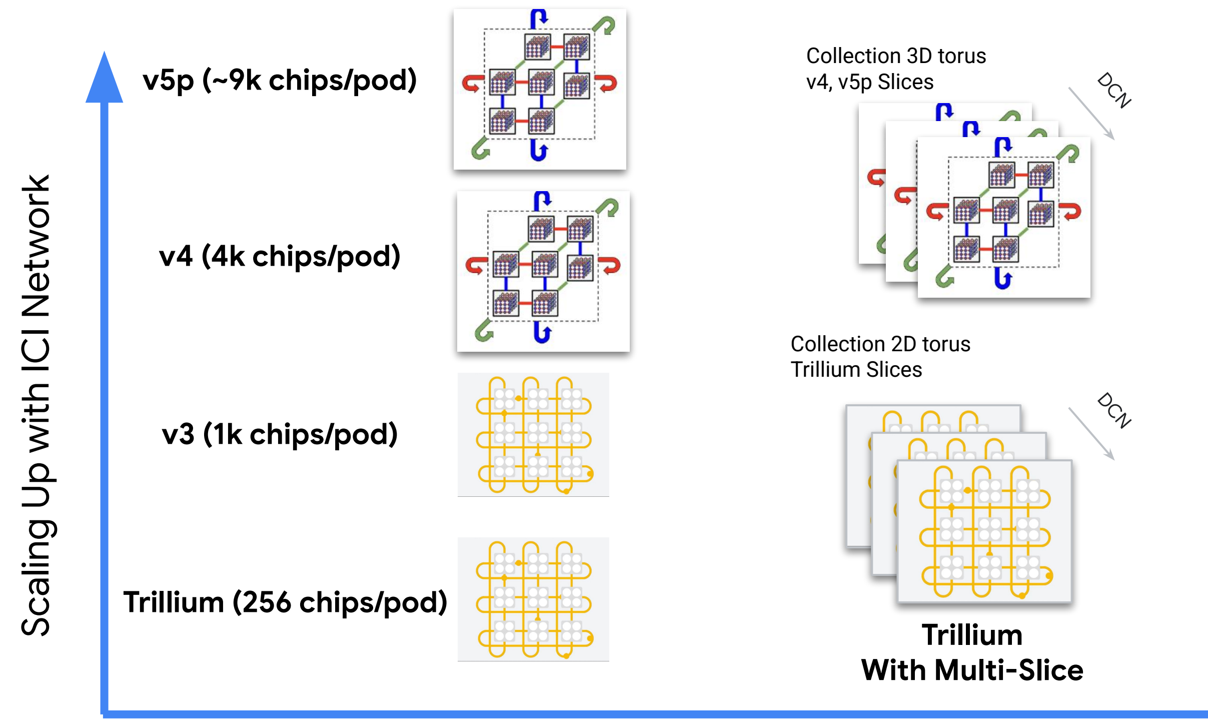




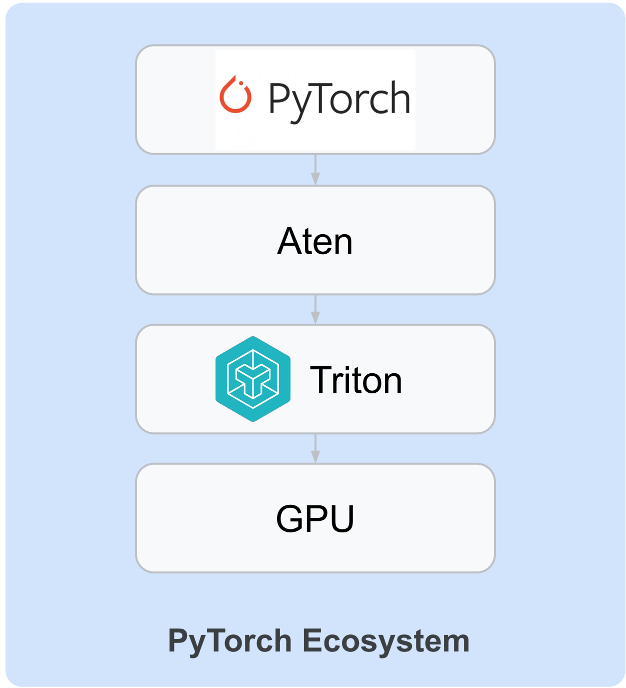
8x growth in cloud TPU chip/hour usage over 12 months

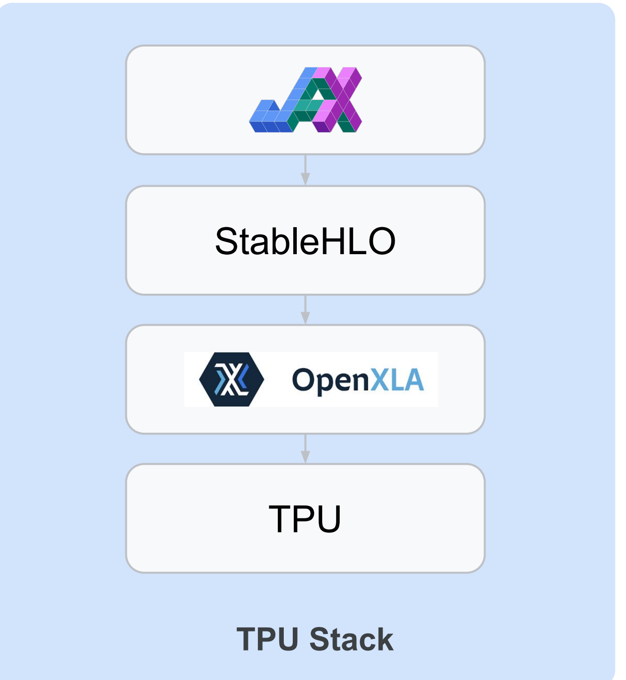
Cloud TPUs Deliver Massive Scale Up & Out

Proprietary + Confidential

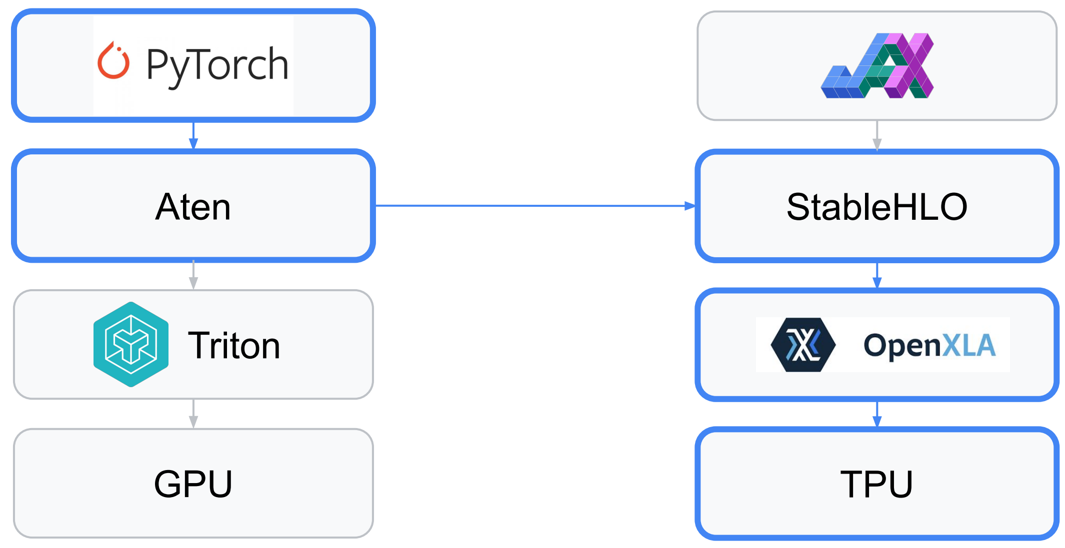


A Tale of Two Stacks





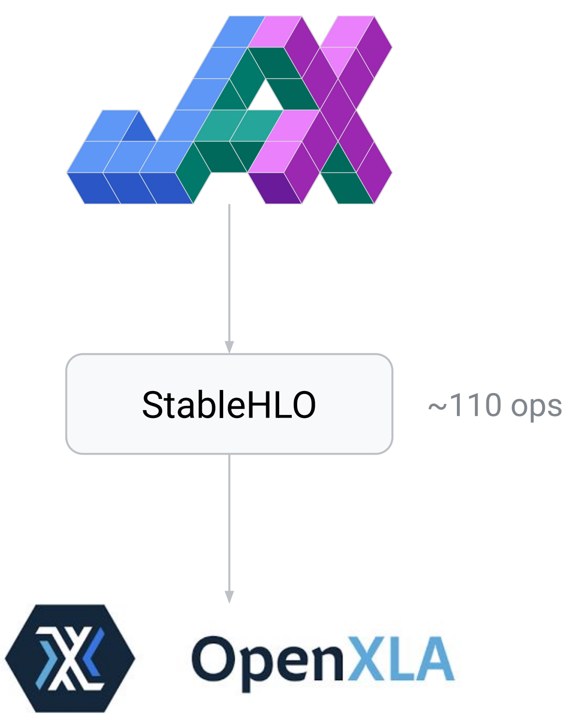
TorchTPU Stack

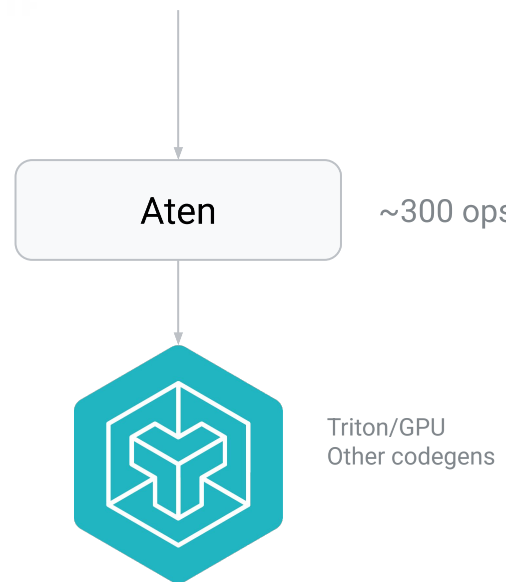


ATen -> SHLO

- Useful abstraction layer for PT users
- Integrates directly into core XLA lowering path

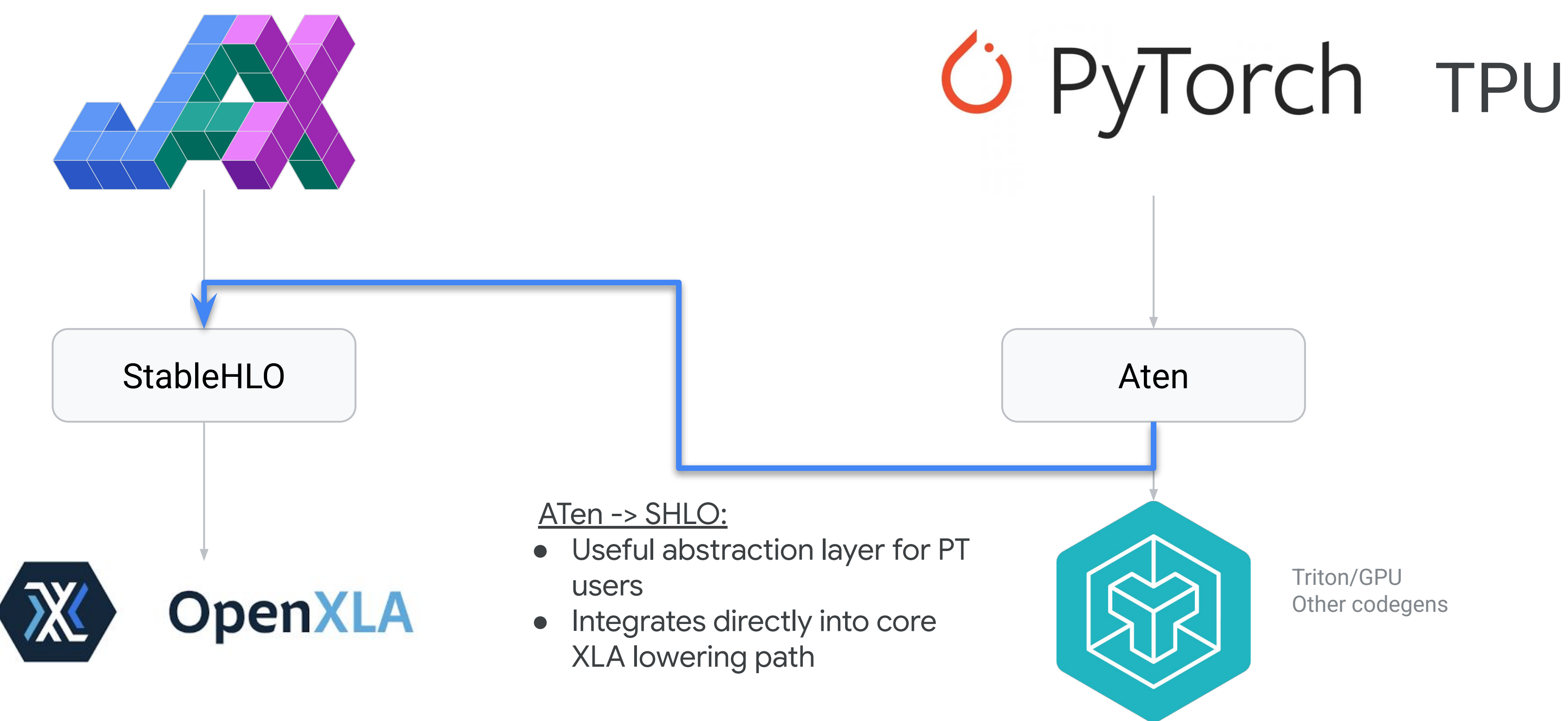
High Level (simplified)





Google

High Level (simplified)



ATen -> SHLO:

- Useful abstraction layer for PT users
- Integrates directly into core XLA lowering path

PyTorch with CUDA backend

```python
import torch
import torch.distributed as dist
from torch.distributed.fsdp import FullyShardedDataParallel as FSDP
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset, DistributedSampler
from transformers import AutoModelForCausalLM

def setup_dist():
 dist.init_process_group(backend="nccl")

def simple_example():  # pylint: disable=missing-function-docstring
 # Distributed device setup
 setup_dist()
 rank = dist.get_rank()
 device = torch.device("cuda", rank)
 torch.cuda.set_device(device)

 model = AutoModelForCausalLM.from_pretrained("meta-llama/Meta-Llama-3-8B")
 model = FSDP(model, device_id=rank, auto_wrap_policy=...)
 optimizer = optim.AdamW(model.parameters(), lr=1e-4)

 # Dataset and DataLoader
 dataset = ...
 sampler = DistributedSampler(dataset, ...)
 dataloader = DataLoader(dataset, ...)

 # Training loop
 num_epochs = ...
 model.train()
 for epoch in range(num_epochs):
   sampler.set_epoch(epoch)
   for batch in dataloader:  # pylint: disable=unused-variable
     # Move batch to device
     input_ids, labels = ...  # pylint: disable=unpacking-non-sequence
     optimizer.zero_grad()
     outputs = model(input_ids=input_ids, labels=labels)
```

PyTorch with TPU backend

```python
import torch
import torch.distributed as dist
from torch.distributed.fsdp import FullyShardedDataParallel as FSDP
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset, DistributedSampler
from transformers import AutoModelForCausalLM

def setup_dist():
 dist.init_process_group(backend="tpu")

def simple_example():  # pylint: disable=missing-function-docstring
 # Distributed device setup
 setup_dist()
 rank = dist.get_rank()
 device = torch.device("tpu", rank)
 torch.tpu.set_device(device)

 model = AutoModelForCausalLM.from_pretrained("meta-llama/Meta-Llama-3-8B")
 model = FSDP(model, device_id=rank, auto_wrap_policy=...)
 optimizer = optim.AdamW(model.parameters(), lr=1e-4)

 # Dataset and DataLoader
 dataset = ...
 sampler = DistributedSampler(dataset, ...)
 dataloader = DataLoader(dataset, ...)

 # Training loop
 num_epochs = ...
 model.train()
 for epoch in range(num_epochs):
   sampler.set_epoch(epoch)
   for batch in dataloader:  # pylint: disable=unused-variable
     # Move batch to device
     input_ids, labels = ...  # pylint: disable=unpacking-non-sequence
     optimizer.zero_grad()
     outputs = model(input_ids=input_ids, labels=labels)
```

Goal: Switching from GPU to TPU requires changing only 3 lines related to device initialization.