



CONFERENCE

—NORTH AMERICA 2026—

From DTensor to spmd_types

Config-based sharding and explicit SPMD in TorchTitan
Chien-Chin Huang · Pian Pawakapan

Distributed training is hard

- Combining parallelisms
 - **Which tensors:** weights and activations
 - **Which layouts:** replicated or sharded, on which dimensions
 - **Which collectives:** all-reduce, all-gather, reduce-scatter, all-to-all
 - Backward needs the right collectives too
- Missing collectives can silently give incorrect loss and gradients
- We need a single program, multiple data (SPMD) description
 - Easy to write and check
 - TorchTitan sharding configs + spmd_types

TorchTitan: from partial DTensor to spmd_types

Before (2025): partial DTensor

TP: DTensor compute. EP: DTensor boundaries, local_map inside.
DP / CP used separate paths.

1. Sharding design

- From partial DTensor to full SPMD
- Config-based module layouts
- Explicit model communication

2. spmd_types

- Forward types fix backward types
- Catch missing collectives or incorrect sharding
- Typecheck; erase types for production

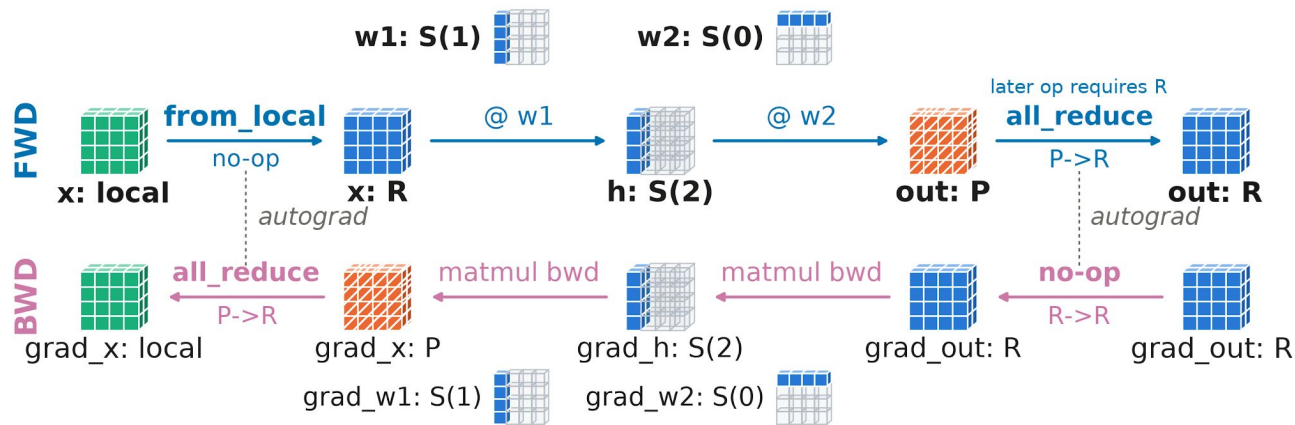
After (2026): full SPMD

Plain tensors + explicit collectives + SPMD typechecking.
FSDP2: DTensor at rest. DCP: raw tensors + its own metadata.

DTensor: placement-based authoring

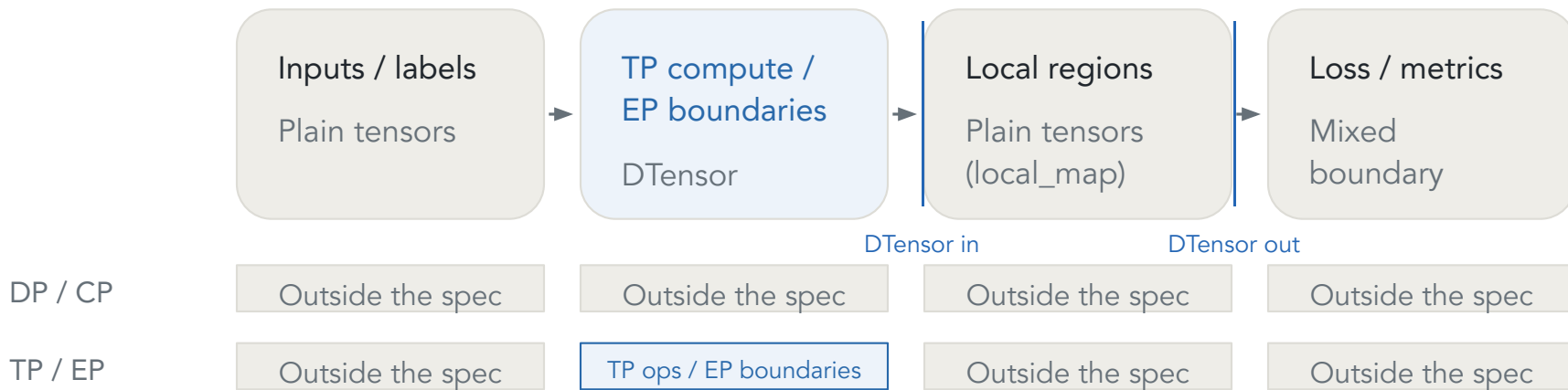
```
# TP MLP (SP off); tp is a 1-D DeviceMesh
# x: [batch, seq, hidden]; w1 and w2 are local shards
x = DTensor.from_local(x, tp, [Replicate()])
w1 = DTensor.from_local(w1, tp, [Shard(1)])
w2 = DTensor.from_local(w2, tp, [Shard(0)])
out = F.silu(x @ w1) @ w2
```

- DTensor infers layouts.
- Communication depends on the mesh and layouts.
- Checkpointing reuses DTensor layouts.



Partial annotation left gaps

- We used DTensor for TP compute and EP boundaries.



Separate CP hooks. Mixed tensor boundaries. Untracked local code.

Our solution: describe the full SPMD program

- Declare each tensor's layout on every relevant compute axis.

Dense compute example, with sequence parallelism enabled

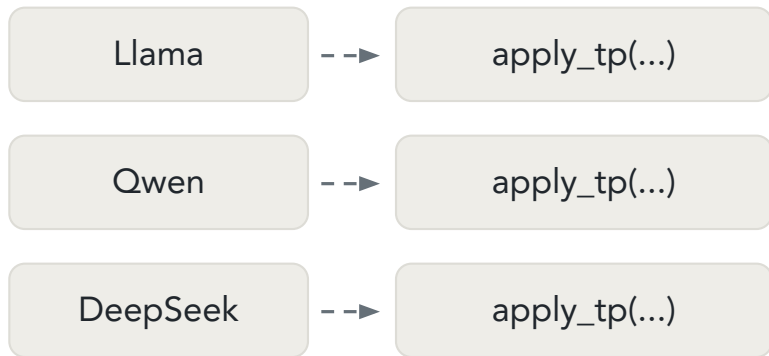
Tensor / state	DP	CP	TP
FFN input	Batch shard	Token shard	Token shard
w13.weight	Replicated	Replicated	Feature shard
Inputs/Labels	Batch shard	Token shard	Replicated

Rank-local regions also declare their SPMD semantics.

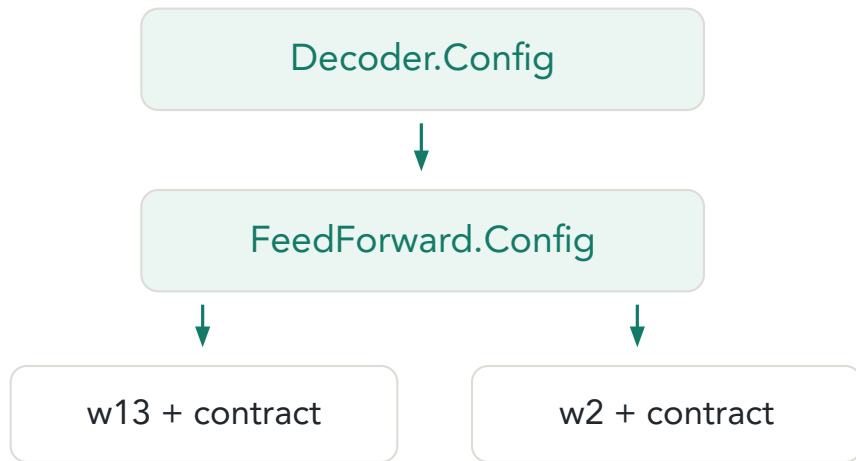
From full SPMD to the config tree

- A complete SPMD contract needs a place to declare module layouts.

Per-model setup



Component-owned config



Reuse sharding. Swap components. Build before materializing.

A module declares its sharding contract

- Config selects the module and declares its distributed interface.

rowwise_sharding_config, simplified

```
def rowwise_sharding_config(output_layout):  
    return ShardingConfig(  
        state_shardings={  
            "weight": dense_param_placement(  
                tp=spmd.S(1)),  
        },  
        in_src_shardings={  
            "input": dense_activation_placement(  
                tp=spmd.S(-1), cp=spmd.S(0)),  
        },  
        out_src_shardings=output_layout,  
    )
```

state_shardings

Parameter and buffer layouts

in_src_shardings / out_src_shardings

Expected inputs and produced outputs

local_spmd (separate option)

A declared rank-local region

Full DTensor: what we learned

- We used full DTensor to verify our config-based sharding design.

Runtime propagation

Per-op dispatch and layout propagation.

Changing shapes can miss the cache.

Implicit communication

Rules can insert redistributions.

Gradient layouts are propagated separately.

Untracked local regions

to_local / local_map do not track local ops.

Boundary and gradient assumptions need care.

Validation results

Config-based sharding preserved numerics and made sharding explicit.
Slow multi-axis planning; CP / MoE compile failures.

TorchTitan today: explicit communication

- Layouts come from config. Modules directly call the collectives.

Config

States + activations

Muse Spark Open recipe:

tinyurl.com/23p44ojs



Module

Local ops + explicit collectives

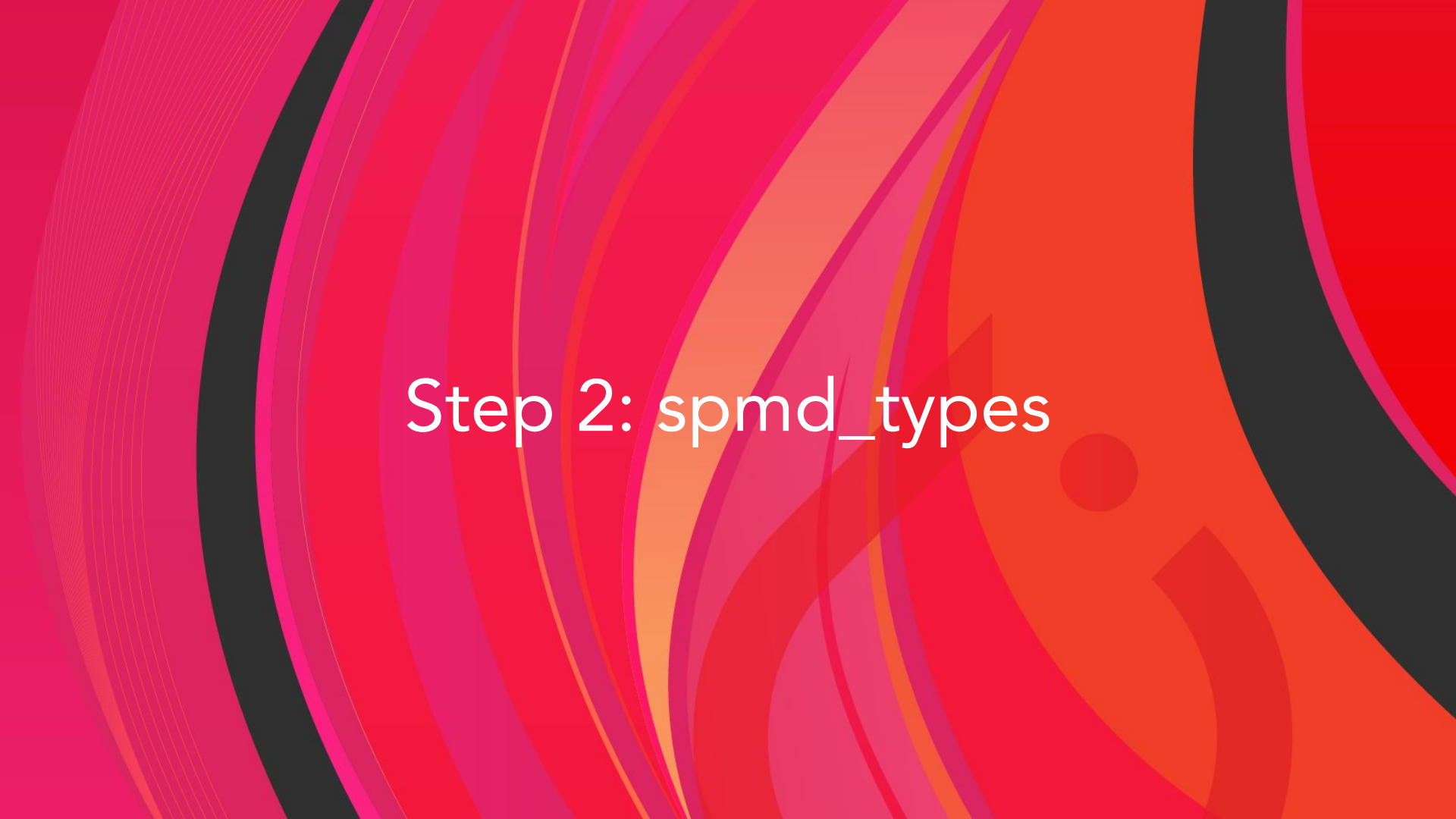
Linear module source:

tinyurl.com/2d8uognp

RowParallelLinear.forward, simplified

```
output = self._linear(input, weight, bias)
return spmd.redistribute(
    output,
    tp_group,
    src=spmd.P,
    dst=output_layout,
)
```

DTensor is still used to supply placement metadata for DCP



Step 2: `spmd_types`

spmd_types

- No implicit collectives
- Tied FWD-BWD typing
- SPMD typechecker for distributed correctness
- Erasure mode for zero runtime overhead
- Local + Global SPMD offerings

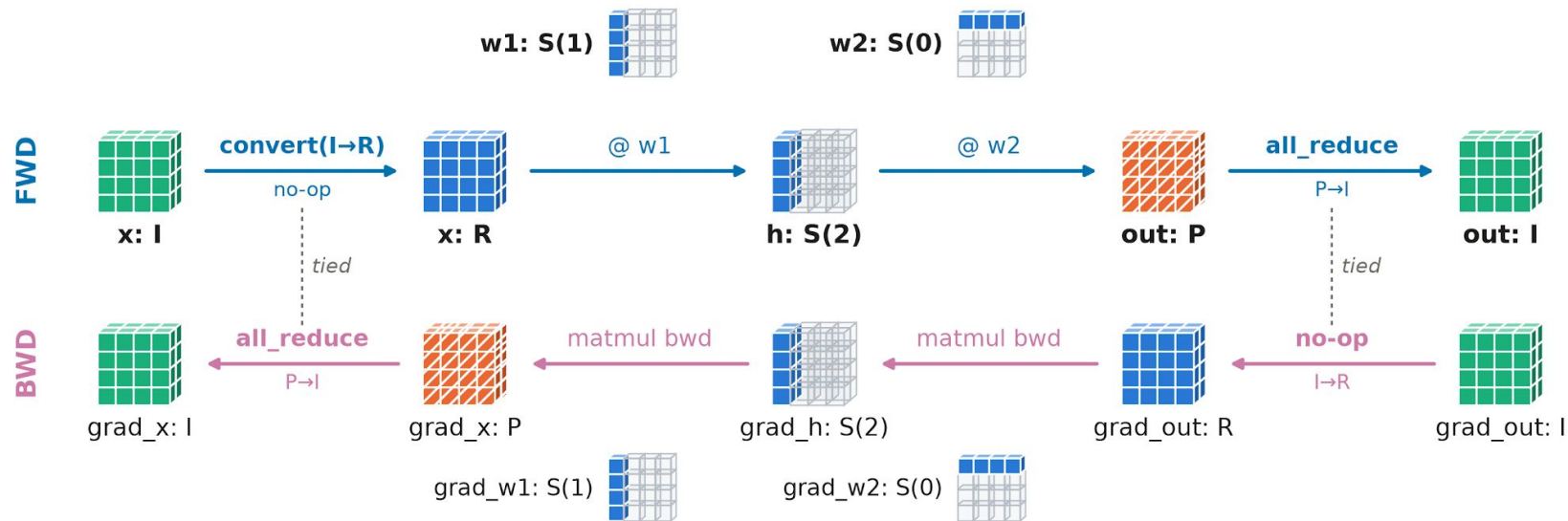
SPMD typechecking: bug-free parallelism

- TP and CP are notorious, collectives closely tied to FWD-BWD compute
 - FSDP, PP, EP are more orchestration-layer / specialized kernels
- Easy to reason about statically, hard to track in an actively evolving training codebase
 - Which activations / gradients are sharded, replicated, or pend reductions
 - Where to insert BWD reductions
- [Megatron-LM #7452](#): 13 silent numerical bugs from one spmd_types-guided audit
- Bugs: missing / extra all-reduces, statistics over local instead of global shard, wrong PG usage

Types

FWD	BWD	Meaning	Example	
R	P	same FWD tensor	FSDP / DDP params after unshard	
I	I	same FWD+BWD compute	TP-replicated params (RMSNorm, router), no SP	
S(i)	S(i)	sharded	TP-sharded weights; DP/CP/SP-sharded activations	
P	R	pending sum	matmul output over a sharded contracting dim	
V	V	varies, no global relation	MoE expert region; multimodal with varying # patches	

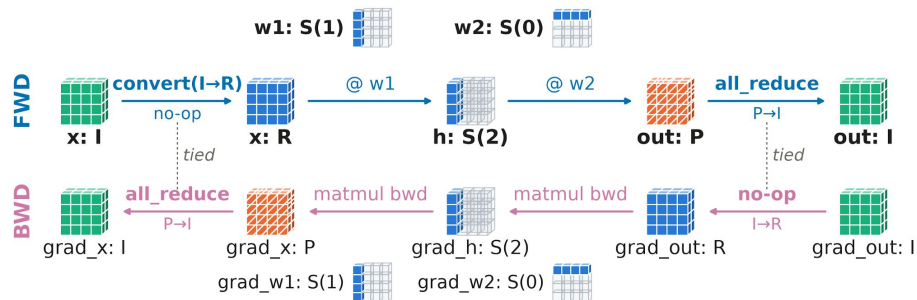
TP MLP, typed (SP off)



SP on (standard): the input all-gathers $S(1) \rightarrow R$ and the output reduce-scatters $P \rightarrow S(1)$.

Placement-based authoring

```
# TP MLP (SP off)
# annotate (checked in dev, erased in prod)
x = spmd.assert_type(x, {tp: spmd.I})
w1 = spmd.assert_type(w1, {tp: spmd.S(1)})
w2 = spmd.assert_type(w2, {tp: spmd.S(0)})
# the only distributed code
x = spmd.redistribute(x, tp, src=spmd.I, dst=spmd.R)
out = F.silu(x @ w1) @ w2
out = spmd.redistribute(out, tp, src=spmd.P, dst=spmd.I)
```

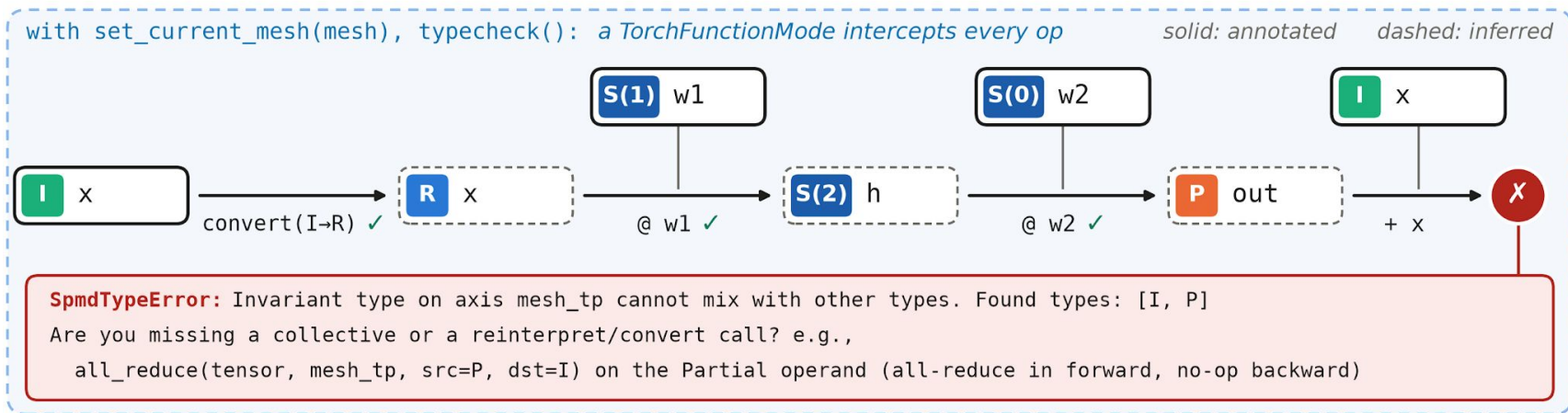


FWD	fwd op	BWD	bwd op
I→R	no-op	P→I	all-reduce
V→R	all-gather	P→V	reduce-scatter
P→V	reduce-scatter	V→R	all-gather
P→R	all-reduce	P→R	all-reduce
V→V	all-to-all	V→V	all-to-all
R→V	shard	V→P	zero-pad
P→I	all-reduce	I→R	no-op
...	...	...	...

For Megatron users: I→R = `copy_to_tensor_model_parallel_region`, P→I = `reduce_from_tensor_model_parallel_region`, ...

The typechecker

- 1 Annotate inputs + params — 2 Set mesh, typechecker on — 3 Propagate types + check



- BWD correctness follows from FWD
- Erasure mode: typechecker off at runtime, zero overhead
- Per-region, per-mesh-axis: global or local SPMD
- Easy cross-mesh transitions: set the ambient mesh, use different PGs

FakeProcessGroup: N-D parallelism on 1 GPU

- Fake PG: a PyTorch ProcessGroup backend whose collectives are no-ops
 - Outputs have the right shapes, but no real comms between ranks
- A TorchTitan fake PG run
 - One process plays one rank of the full job, e.g. rank 0 of 8192 GPUs
 - Builds the full mesh, shards the model, runs a real training step
 - Shapes, sharding, and per-rank GPU memory are real; outputs, collective durations are not
- Correctness for large-scale training, using 1 GPU
 - SPMD typechecking needs ops, rules, input types, not real tensor values
 - Check sharding annotations, collectives, reductions, memory usage, before scaling to real cluster

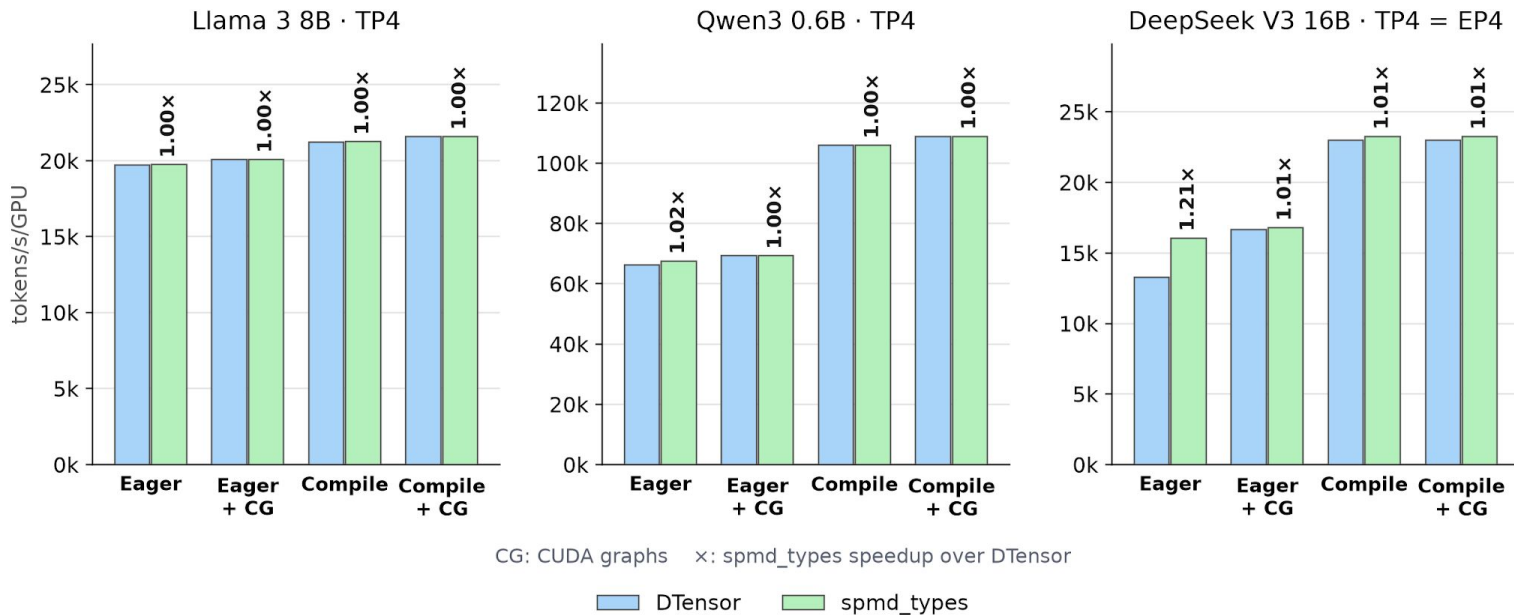


TorchTitan today

TorchTitan today

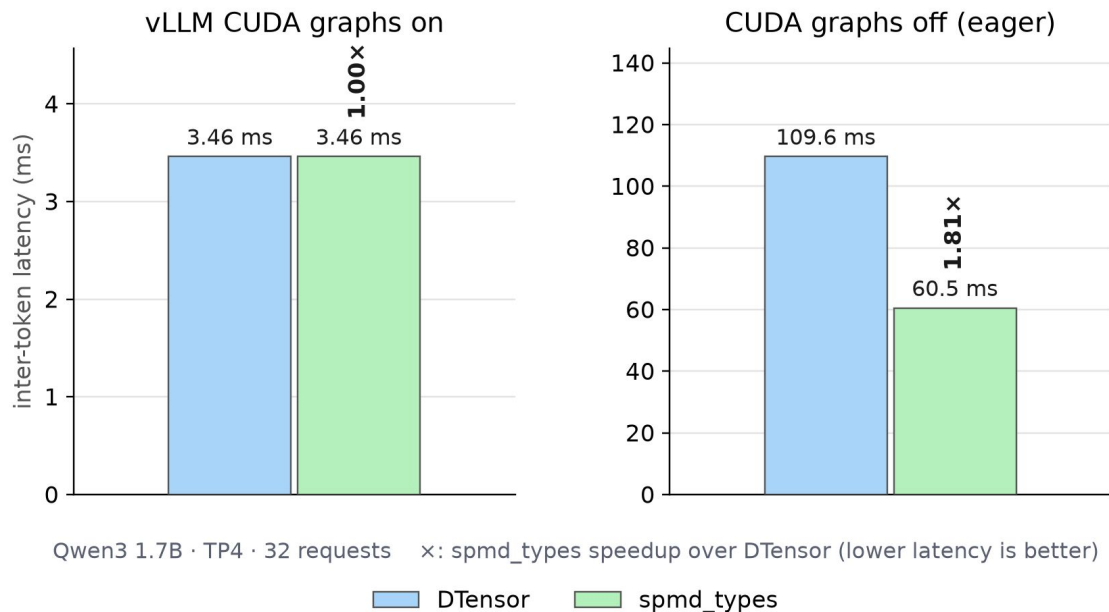
- Replaced the DTensor backend: all FWD-BWD compute runs on `spmd_types`
- Typechecking support for all models and parallelisms
 - Models: Llama 3, Qwen3 / 3.5 / 3.6 / 3.8, DeepSeek V3 / V4, GPT-OSS, Kimi K2.7 / K3, Muse Glimmer, Flux
 - Parallelisms: FSDP / HSDP / DDP, TP + SP, CP (all-gather KV, Ulysses), EP (all-to-all, DeepEP, HybridEP)
 - Fake PG support built-in
- TorchTitan feature composability
 - TitanRL (trainer + generator), GraphTrainer, SimpleFSDP
 - PP, torch.compile, CUDA graphs, MXFP8 / NVFP4, fused kernels, AC, DCP
 - Full typechecking support for all features still WIP
- Bitwise-equivalent to the DTensor backend

Training perf: spmd_types vs DTensor



Llama3 / Qwen3 : 4× GB300, no activation checkpointing, steady-state TPS. DeepSeek V3: HybridEP, selective AC

RL decode perf: spmd_types vs DTensor



Standalone vLLM generator running the TitanRL path, 4x GB300, TP4, 32 requests, 256 prompt / 256 output tokens, median decode step. CUDA graphs off = vLLM enforce_eager, no compile.

Ongoing directions

- DTensor at rest, spmd_types in compute
 - DTensor follows FSDP2 for storage-time params, grads; provides DCP interface
- TorchTitan: migration towards explicit collectives
 - Eliminating src->dst config-based collectives, write in forward for easy extensibility

Conclusion

- `spmd_types` makes distributed compute explicit and checkable
 - Forward types determine gradient types
 - Each forward collective fixes its backward
 - Typechecker + fake PG: catch distributed bugs on 1 GPU
 - Erase types in production: plain tensors + explicit collectives
- Sharding configs declare layouts across parallelisms
- TorchTitan: same numerics, lower host overhead
 - DTensor at rest; `spmd_types` in compute
- Code: github.com/meta-pytorch/spmd_types

Acknowledgments

Tianyu Liu, Edward Z. Yang, Ailing Zhang, Pei Zhang, Vishal Nandavanam

- For design, spmd_types library, reviews



Q&A