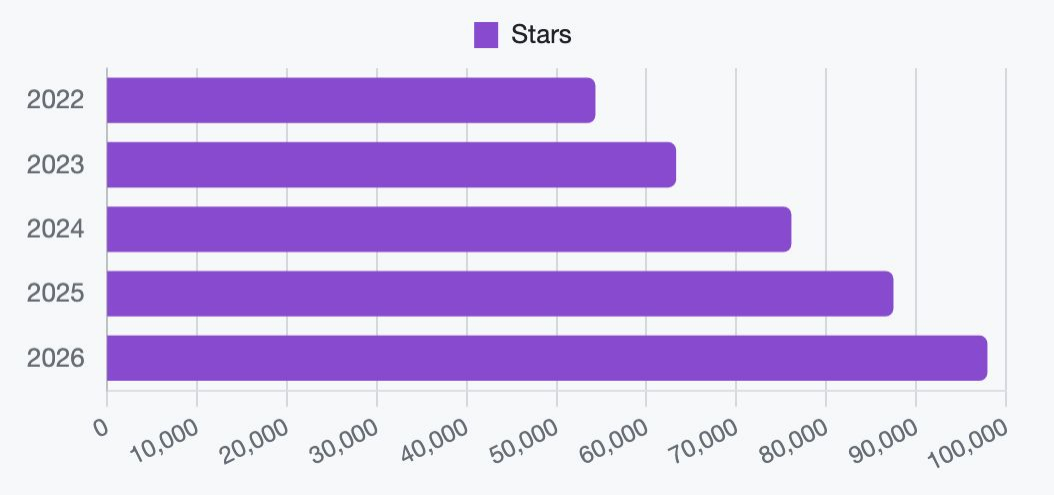
PyTorch Project Update

PyTorch Conference Europe 2026

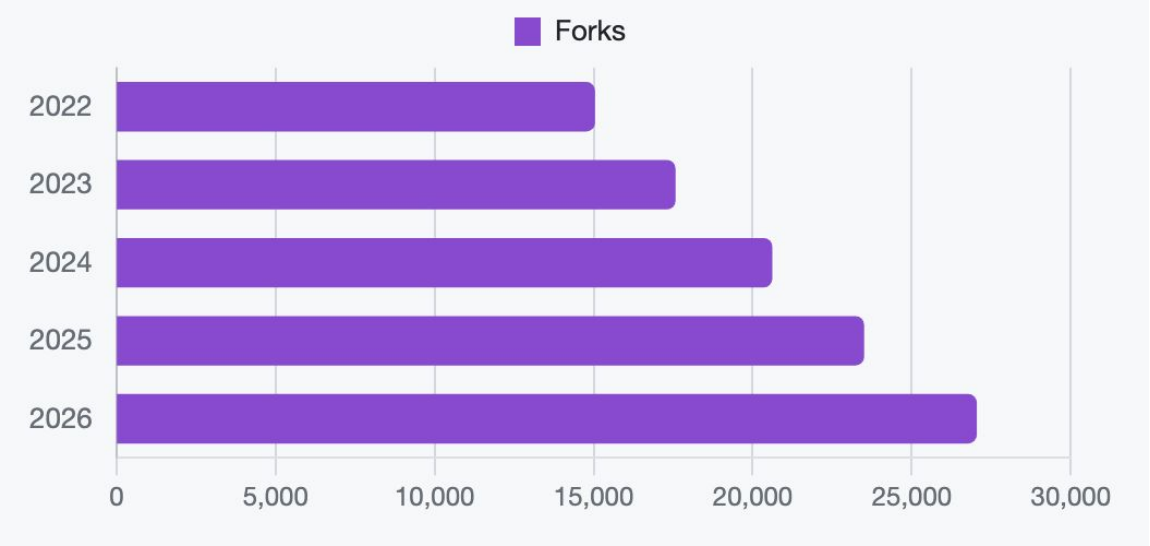

PyTorch
CONFERENCE PARIS

Stars



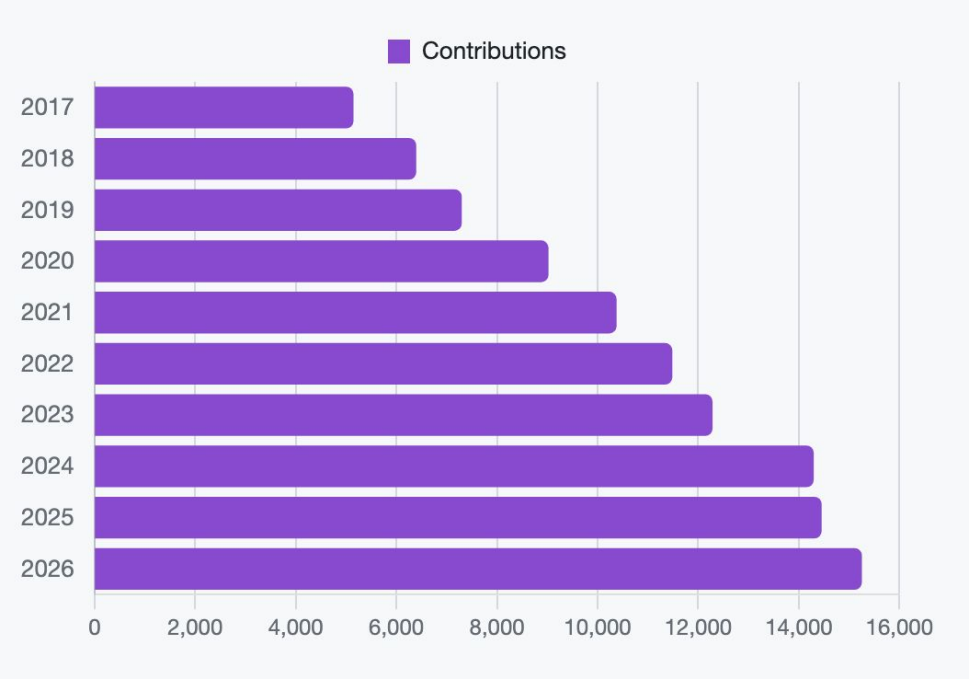
Approaching 100K stars

Forks



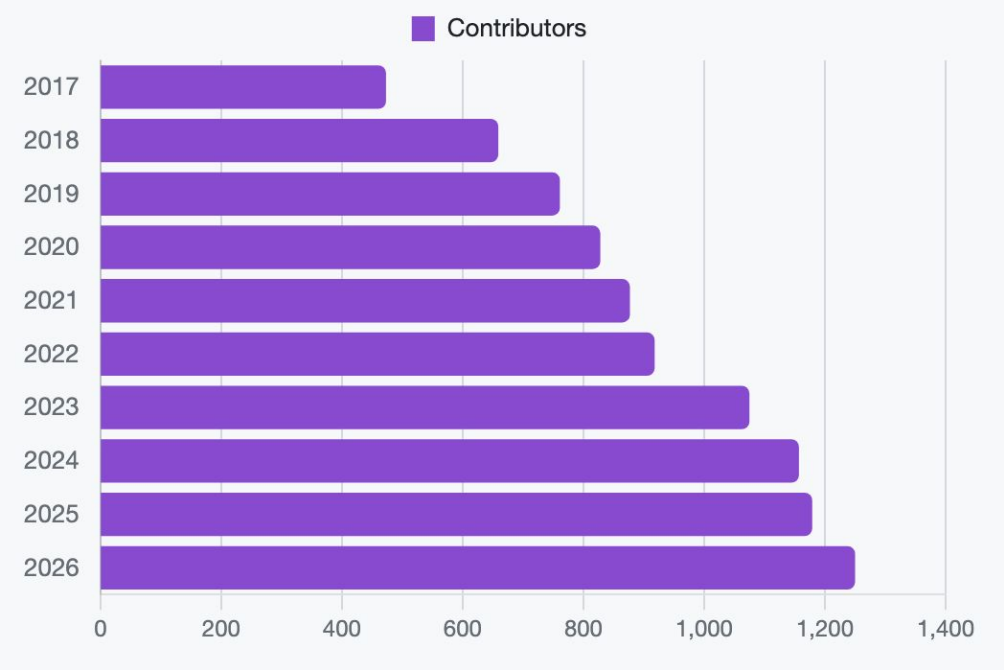
27K forks, nearly doubled from 2022

Contributions



> 15K
Contributions made to pytorch core library as of 2026

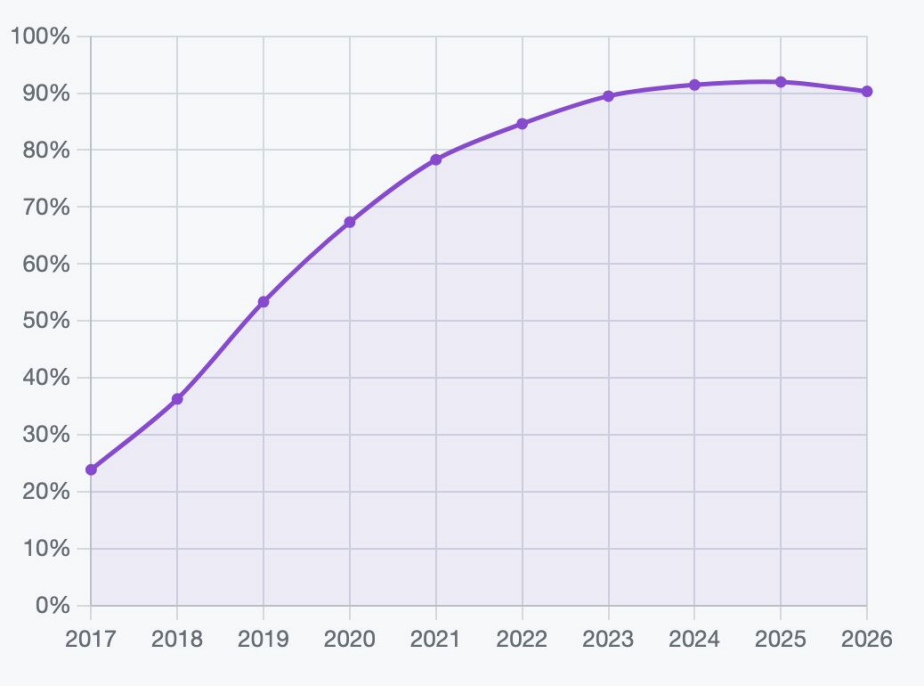
Contributors



1250

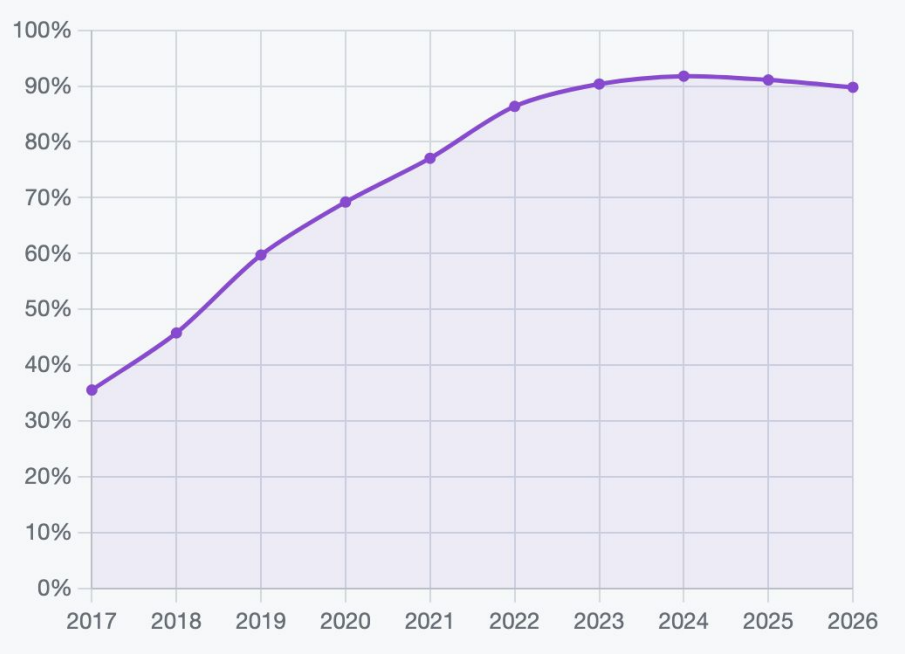
New contributors joined our community as of the last 12 months!

Adopted by > 90% of research projects



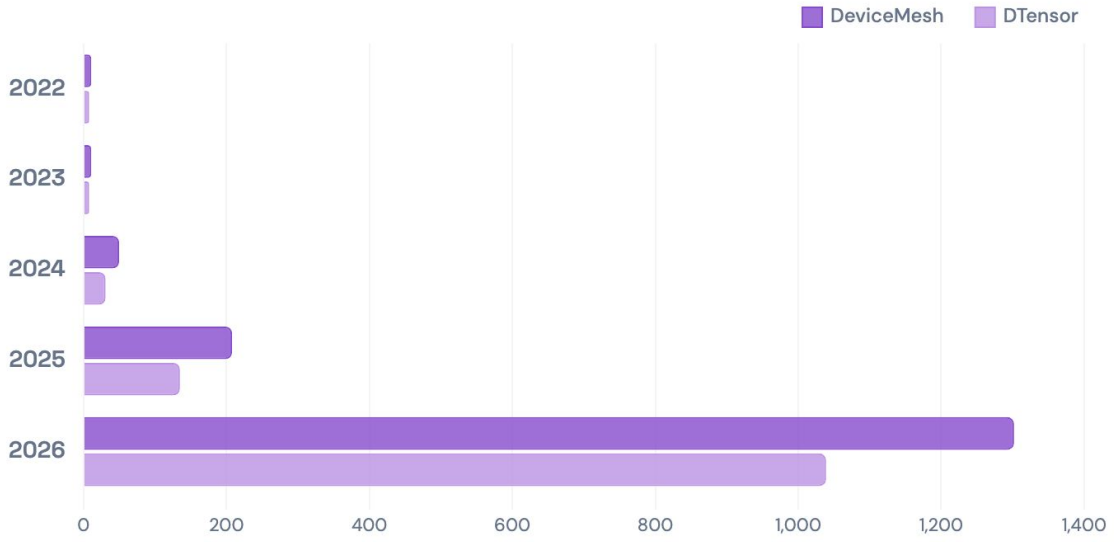
9 in 10 research projects adopt PyTorch

Strong hold among open source projects



~90% of open-source ML projects are using PyTorch

DeviceMesh and DTensor continue to rise



Both are leveraged by over 1K open source AI projects in the last 12 months, e.g.,

thu-ml/TurboDiffusion

Robbyant/lingbot-vla

OpenMOSS/MOVA

What's new in PyTorch?

In this keynote:

- Pre-compilation support
- SPMD types

Come chat with us at Meet the Maintainers (10:30 CEST today!)

In this conference:

- Helion
- Executorch
- Compile
  - Combo kernels
  - Flash Attention v4 for FlexAttention
  - CUDA streams
  - Pipeline parallelism
  - C++ wrapper
  - Custom op autotuning
- TorchTPU
- Monarch and TorchStore
- Stable ABI
- FP8

Updates from inside PyTorch

- Precompile (could be dedicated slide)
- Opaque objects
- invoke_subgraph (Dynamo)
- CuteDSL in core
- Single-dim shard prop
- Streams in compile
- DebugMode

Pre-compilation support for torch.compile

torch.compile in distributed training jobs: the status quo

- Each rank performs compilation *independently*
- Implicit requirement: every rank (re)compiles in exactly the same way
- Problems:
  - It's slow (Will the cache hit? How fast is Dynamo?)
  - It's complicated (Dynamic shapes--the graphs/recompile patterns may not be the same.)
  - It's unsafe (Recompiles vs NCCL timeouts; collective reordering/bucketing dangeorus.)

Solution: Pre-compilation

- Pre-compile on a single GPU (or even CPU!)
- Distribute compiled artifact to all ranks
- Much simpler! Guaranteed no compile, guaranteed no divergence.

Two problems to solve:

- Make it possible to pre-compile the model at all (e.g., full model capture)
- Ensure that a single artifact works for all ranks (“compile on one rank”)

What have we got? Preview support for a “graph trainer” in torchtitan, supporting llama3.

What it looks like

```
# Step 1: precompile on a single process (needs only 1 GPU)
python -m torchtitan.experiments.graph_trainer.precompile_main \
    --module graph_trainer.llama3 \
    --config graph_trainer_llama3_debugmodel \
    --compile.passes full_inductor_compilation \
    --compile.joint_passes inductor_decomposition \
    --compile.precompile_artifact_dir /tmp/precompile_artifacts \
    --parallelism.data_parallel_shard_degree 4 \
    --parallelism.tensor_parallel_degree 2
```

What it looks like

```
# Step 2: load and train with torchrun (uses all GPUs)
torchrun --nproc_per_node=8 --virtual-local-rank
    -m torchtitan.train \
    --module graph_trainer.llama3 \
    --config graph_trainer_llama3_debugmodel \
    --compile.passes full_inductor_compilation \
    --compile.joint_passes inductor_decomposition \
    --compile.precompile_artifact_dir /tmp/precompile_artifacts \
    --parallelism.data_parallel_shard_degree 4 \
    --parallelism.tensor_parallel_degree 2
```

What's next

- Covering more models in torchtitan with pre-compile support
- Support uneven sharding across ranks
- Handling for intentional non-SPMD'ness (e.g., logging on only one rank)

Still quite early! Feedback and suggestions welcome!

Check it out in torchtitan (e.g., https://github.com/pytorch/torchtitan/pull/2713)

SPMD types

What are SPMD types?

Single Program, Multiple Data

A lightweight way of writing down properties of a program, that can be mechanically checked for correctness

A type system for describing the distribution of tensors in SPMD programs.

- Is the value the same or different across the ranks?
- Does the value (or its gradient) represent a pending reduction, i.e., that you should do an all-reduce across ranks to find the real value?

We call this local SPMD (since we don't say how to assemble a full tensor)

What problem do SPMD types solve?

The PyTorch guarantee: if you can write the forwards, we can differentiate it!*

*But not for collectives...

Why no differentiable collectives?

You can do it, and Megatron-like training frameworks have custom autograd functions that make tensor parallelism work.

Problem: You have lots of weird extra no-op operations which need to be called in the right places (e.g., f, is a no-op in forwards, all-reduce in backwards). If you get it wrong, silently incorrect derivatives!

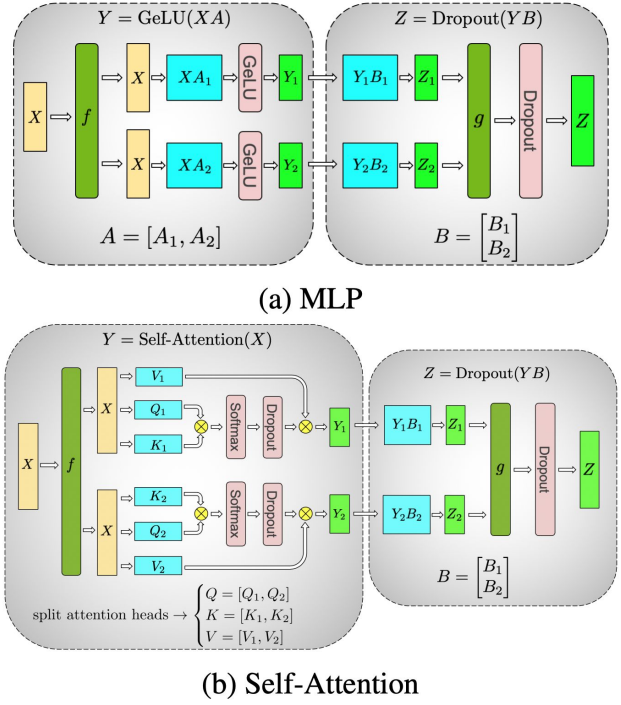


Figure 3. Blocks of Transformer with Model Parallelism. f and g are conjugate. f is an identity operator in the forward pass and all reduce in the backward pass while g is an all reduce in the forward pass and identity in the backward pass.

Why no differentiable collectives?

Argument: LLM parallelism recipe is pretty well understood, practitioners “know” where they need to put special autograd functions, just learn it and reapply it.

Counter-argument:

Why not DTensor?

Write code as if operating on a global tensors (with traditional autograd rules), DTensor will automatically insert collectives in forwards and backwards to allow for sharding.

Trouble:

- Some computation is more naturally expressed on local SPMD basis (local_map has the same problem as described above!)
- Implicit collective insertion leads to unexpected communications
- DTensor eager overhead is very high!

Local SPMD types: ⚠️TRADE OFFER⚠️

I receive: Your plain tensor Megatron-style code with explicit collectives, but with extra type annotations on your input and weight tensors

You receive:

- A guarantee that well-typed programs always have correct derivatives
- Type errors that tell you where you are missing collectives: a way of defining and enforcing parallelism related contracts across module boundaries

...without any runtime overhead, while supporting all of the communication patterns you are doing today! (And it works out of the box with torch.compile!)

Related work: JAX sharding-in-types

- It's the same thing, if you use the undocumented reduced/unreduced types. https://blog.ezyang.com/2026/01/jax-sharding-type-system/
- Huge credit to the JAX team for convincing us that backward sharding should be strictly derivable from forward sharding (not true with DTensor today!)
- We've adapted it for the PyTorch ecosystem and changed some defaults

What is a local SPMD type

For each axis on your mesh, the local SPMD type answers three questions:

- Is the value the different across the ranks? (Varying)
- Does the value need an reduction (e.g., all-reduce) across the ranks? (Partial)
- Does the gradient need a reduction across ranks or not? (Replicate vs Invariant)

[TODO EDIT]

Local SPMD types: Varying

```
Rank 0   [ 1.0, 0.1 ]
Rank 1   [ 2.0, 0.2 ]
Rank 2   [ 3.0, 0.3 ]
```

Compare with DTensor Shard placement: DTensor Shard says how to concat the ranks together to form a full tensor; Varying says they vary, but doesn't say how the rank local tensors relate to each other.

Local SPMD types: Partial

```
Rank 0   +[ 1.0, 0.1 ]
Rank 1   +[ 2.0, 0.2 ]  -- all_reduce -->  [ 6.0, 0.6 ]
Rank 2   +[ 3.0, 0.3 ]
```

You are *committed* to doing the all-reduce: the type system won't let you forget!
(E.g., it's illegal to do a local, non-linear operation on a partial tensor)

Local SPMD types: Invariant

```
Rank 0   [ 1.0, 0.1 ]          # Forwards
Rank 1      - “ -
Rank 2      - “ -
```

Same value on every rank... and gradient is same value on every rank too!
(Compare with DTensor Replicate, which doesn’t say anything about backwards.)

```
Rank 0   [ 0.5, 0.3 ]          # Backwards
Rank 1      - “ -
Rank 2      - “ -
```

Local SPMD types: Replicate

```
Rank 0   [ 1.0, 0.1 ]          # Forwards
Rank 1      - “ -
Rank 2      - “ -
```

Same value on every rank... but the backwards is Partial! (It turns out, most typical use of Replicate DTensor has Partial backwards, so used this name here.)

```
Rank 0   +[ 0.1, 0.1 ]         # Backwards
Rank 1   +[ 0.0, 0.2 ]
Rank 2   +[ 0.4, 0.0 ]
```

Putting it together

For every axis of your device mesh (e.g., {DP, CP, TP}), type it with a local SPMD type. For example:

$$\{DP: V, CP: V, TP: I\}$$

This means your tensor varies over the DP and CP axes of your mesh, but is invariant (gradients are also is the same) over the TP axis.

Example: DDP/FSDP

```
def forward(self, x):
    spmd.assert_type(x,           {DP: V, CP: V})
    spmd.assert_type(self.weight, {DP: R, CP: R})

    out = F.linear(x, self.weight)
    spmd.assert_type(out,         {DP: V, CP: V})

    return out
```

Recall: Gradient for self.weight is Partial! FSDP handles the all-reduce

DP is pretty boring, CP is boring except when it's not (attention!)

Example: RMSNorm (no sequence parallel)

```
def forward(self, input: torch.Tensor) -> torch.Tensor:
    spmd.assert_type(input,       {TP: I})
    spmd.assert_type(self.weight, {TP: I})

    out = rms_norm(input, self.weight)
    spmd.assert_type(out,         {TP: I})

    return out
```

No sequence parallel; every rank is doing the exact same compute (forwards and backwards)

Example: RMSNorm (sequence parallel)

```
def forward(self, input: torch.Tensor) -> torch.Tensor:
    spmd.assert_type(input,       {TP: V})    <- input is now sharded on TP
    spmd.assert_type(self.weight, {TP: I})

    out = rms_norm(input, self.weight)
    spmd.assert_type(out,         {TP: V})

    return out
```

Example: RMSNorm (sequence parallel)

```
def forwar        inp        ch.Tensor) -> torch.Tensor:
    spmd.a                          {TP: V})     <- input is now sharded on TP
    spmd.asse                 .weight, {TP: I})

    out = r                   elf.weight)
    spmd.a                    ype(          {TP: V})

    return out
```

⚠️ Invariant type on axis TP cannot mix with other types. Found types: [I, V]

Example: RMSNorm (sequence parallel)

```
SpmdTypeError: Invariant type on axis TP cannot mix with other types. Found
types: [V, I]
  Are you missing a collective or a reinterpret/convert call? e.g.,
    invariant_to_replicate(tensor, TP) on the Invariant operand

    In rms_norm(
      input: f32[2, 4] {TP: V},
      normalized_shape: (4,),
      weight: f32[4] {TP: I},
      eps: 1e-05,
    )
```

Example: RMSNorm (sequence parallel)

```
def forward(self, input: torch.Tensor) -> torch.Tensor:
    spmd.assert_type(input,       {TP: V})
    spmd.assert_type(self.weight, {TP: R})

    out = rms_norm(input, self.weight)
    spmd.assert_type(out,         {TP: V})

    return out
```

Recall: Gradient for self.weight is Partial! Someone else needs to take care of the all-reduce.

THE RULE: Replicate, Varying can mix.
Invariant can only mix with itself.

Example: RMSNorm (sequence parallel)

```
def forward(self, input: torch.Tensor) -> torch.Tensor:
    spmd.assert_type(input,       {TP: V})
    spmd.assert_type(self.weight, {TP: I})

    weight = spmd.invariant_to_replicate(self.weight, TP)
    spmd.assert_type(weight, {TP: R})

    out = rms_norm(input, weight)
    spmd.assert_type(out,         {TP: V})

    return out
```

Alternative: convert Invariant to Replicate (forward no-op; backward all-reduce). No more pending all-reduce on weight!

Example: Linear

	No sequence parallel	Sequence parallel

Col-parallel
(Linear produces Varying)
	`input: {TP: I}`
	`linear_out: {TP: V}`
	`out: {TP: V}`
		`input: {TP: V}`
		`linear_out: {TP: V}`
		`out: {TP: V}`

Row-parallel
(Linear produces Partial)
	`input: {TP: V}`
	`linear_out: {TP: P}`
	`out: {TP: I}`
		`input: {TP: V}`
		`linear_out: {TP: P}`
		`out: {TP: V}`

Example: Col-parallel Linear (no sequence parallel)

```
def forward(self, x):
    spmd.assert_type(x,           {TP: I})
    spmd.assert_type(self.weight, {TP: V})

    x = spmd.invariant_to_replicate(x, TP)
    spmd.assert_type(x,           {TP: R})

    out = F.linear(x, self.weight)
    spmd.assert_type(out,         {TP: V})

    return out
```

<- this could also be R
(delaying the all-reduce)

Example: Col-parallel Linear (sequence parallel)

```
def forward(self, x):
    spmd.assert_type(x,           {TP: V})
    spmd.assert_type(self.weight, {TP: V})

    x = spmd.all_gather(x, TP, src=spmd.S(seq_dim), dst=R)
    spmd.assert_type(x,           {TP: R})

    out = F.linear(x, self.weight)
    spmd.assert_type(out,         {TP: V})

    return out
```

spmd.S here is suggestive of “sharded on seq_dim”; in local SPMD it just says what tensor dim to cat on.

Example: Row-parallel Linear (no sequence parallel)

```
def forward(self, x):
    spmd.assert_type(x,           {TP: V})
    spmd.assert_type(self.weight, {TP: V})

    out = F.linear(x, self.weight)
    spmd.assert_type(out,         {TP: P})

    out = spmd.all_reduce(out, TP, dst=I)
    spmd.assert_type(out,         {TP: I})

    return out
```

<- this could also be R (downstream delayed the all-reduce)

Example: Row-parallel Linear (sequence parallel)

```
def forward(self, x):
    spmd.assert_type(x,           {TP: V})
    spmd.assert_type(self.weight, {TP: V})

    out = F.linear(x, self.weight)
    spmd.assert_type(out,         {TP: P})

    out = spmd.reduce_scatter(out, TP, dst=spmd.S(seq_dim))
    spmd.assert_type(out,         {TP: V})

    return out
```

Example: (Naive) Mixture of Experts

These are equivalent!

```
def forward(hidden, topk, routes):                                  # {DP: V, CP: V, TP: V}
  x, topk, token_ids = prepare_activations(hidden, topk)
  with spmd.set_current_mesh({EDP, EP, ETP}):                       # {EDP: V, EP: V, ETP: V}
    global_counts = spmd.all_gather(routes.sum(dim=0), ETP, S(0), R) # {EDP: V, EP: V, ETP: R}
    global_counts = spmd.all_gather(global_counts, EP, S(0), R)     # {EDP: V, EP: R, ETP: R}
    ep_is, ep_os, etps = derive_splits(global_counts)
    x = spmd.all_to_all(x, EP, S(0), S(0), ep_is, ep_os)            # {EDP: V, EP: V, ETP: V}
    x = spmd.all_gather(x, ETP, S(0), R, split_sizes=etps)          # {EDP: V, EP: V, ETP: R}
    x = expert_fn(x)                                                # {EDP: V, EP: V, ETP: P}
    x = spmd.reduce_scatter(x, ETP, P, S(0), split_sizes=etps)      # {EDP: V, EP: V, ETP: V}
    x = spmd.all_to_all(x, EP, S(0), S(0), ep_os, ep_is)            # {EDP: V, EP: V, ETP: V}
  return unpermute_and_combine(x, topk, token_ids, hidden)          # {DP: V, CP: V, TP: V}
```

spmd_types API

- High level API
  - spmd.assert_type(x, {axis: spmd.R | spmd.I | spmd.V | spmd.P})
  - with spmd.typecheck(): ...
- All the usual collectives
  - all_gather(x, axis, dst=spmd.V | spmd.S(i), split_sizes=None) # stack or concat on dim i
  - all_reduce(x, axis, dst=spmd.R | spmd.I) # reduced/unreduced grad_in
  - reduce_scatter(x, axis, scatter_dim=None, split_size=None)
  - all_to_all(x, axis, split_dim, concat_dim, ...)
- Special local operations
  - convert(x, axis, src, dst) # semantics preserving conversion; CAN compute
    - invariant_to_replicate(x, axis) = convert(x, axis, src=spmd.I, dst=spmd.R) # fwd is no-op
  - reinterpret(x, axis, src, dst) # coercion from one spmd type to another; NEVER computes

Each of these is a function in Megatron!

```
spmd.convert        (x, tp, src=spmd.I,      dst=spmd.R)
spmd.all_reduce     (x, tp,                  dst=spmd.I)
spmd.reduce_scatter(x, tp,                   dst=spmd.S(-1))
spmd.all_gather     (x, tp, src=spmd.S(-1),  dst=spmd.R)
spmd.convert        (x, tp, src=spmd.I,      dst=spmd.S(0))
spmd.all_gather     (x, tp, src=spmd.S(0),   dst=spmd.R)
spmd.all_gather     (x, tp, src=spmd.S(0),   dst=spmd.I)
spmd.reduce_scatter(x, tp,                   dst=spmd.S(0))
spmd.all_gather     (x, tp, src=spmd.S(-1),  dst=spmd.R)
spmd.reduce_scatter(x, tp,                   dst=spmd.S(-1))
spmd.all_to_all     (x, group)
```

Forwards/Backwards

Fwd Type	Forward	Bwd Type	Backward
R -> V	convert(R,V)	V -> P	convert(V,P)
R -> P	convert(R,P)	R -> P	convert(R,P)
I -> R	convert(I,R)	P -> I	all_reduce(I)
V -> R	all_gather(R)	P -> V	reduce_scatter()
V -> V	all_to_all()	V -> V	all_to_all()
P -> R	all_reduce(R)	P -> R	all_reduce(R)
P -> V	reduce_scatter()	V -> R	all_gather(R)
V -> P	reinterpret(V,P)	R -> V	reinterpret(R,V)

Future work

- Global SPMD types (ala DTensor)
  - All the same shard propagation rules from DTensor
  - Supports “erasure”: no runtime overhead! More explicit API than conventional DTensor
- Single mesh-dim sharding strategies (14:45 CEST today)
- FlexShard
  - SPMD types are *compute* oriented description of how data is distributed, but things like FSDP also need to talk a *storage* oriented description that can be much more complicated. Factoring these separately helps!

Check out spmd_types at https://github.com/meta-pytorch/spmd_types