Scaling Up:

Sharding and Parallelism with JAX and Flax NNX

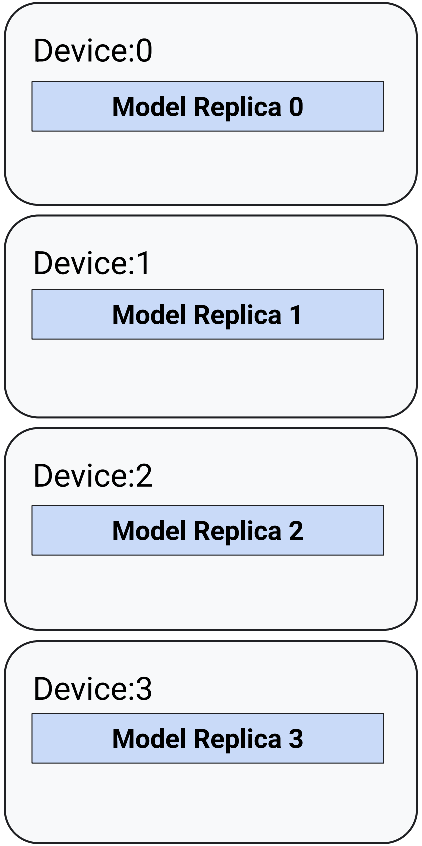
Leveraging Explicit Sharding for Distributed Training

Why Distributed Training?

- **Model Scale**: Models now have billions or trillions of parameters, exceeding single GPU/TPU memory.
- **Data Scale**: Training datasets are massive, requiring parallel processing.
- **Faster Training**: Distributing computation across many devices significantly reduces training time.
- **The JAX Approach**: SPMD: Write a Single Program, let JAX/XLA compile it to run on Multiple Data shards across devices.

Distributed Data Parallelism

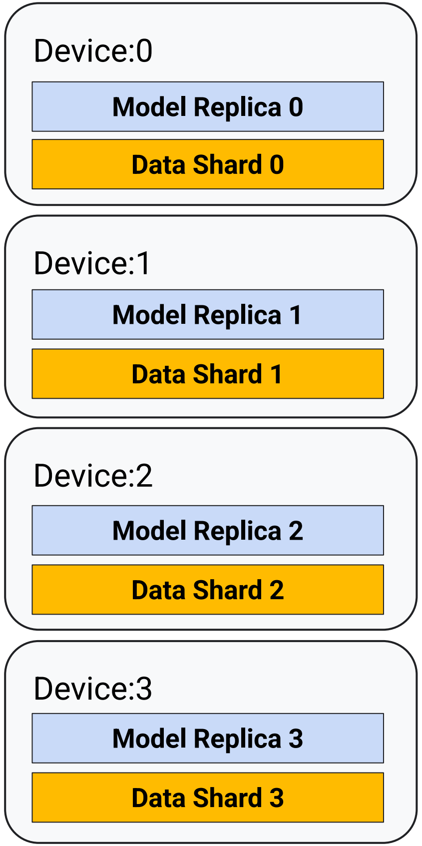
- **Model Replicated, Data Split**: Model copied to multiple devices (GPUs/TPUs); dataset divided into unique batches per device.
- **Parallel Gradient Calculation**: Each device computes gradients independently on its local data batch using its model copy.
- **Gradient Sync & Consistent Update**: Gradients aggregated across devices (e.g., averaged); combined result updates all model copies identically.



Google

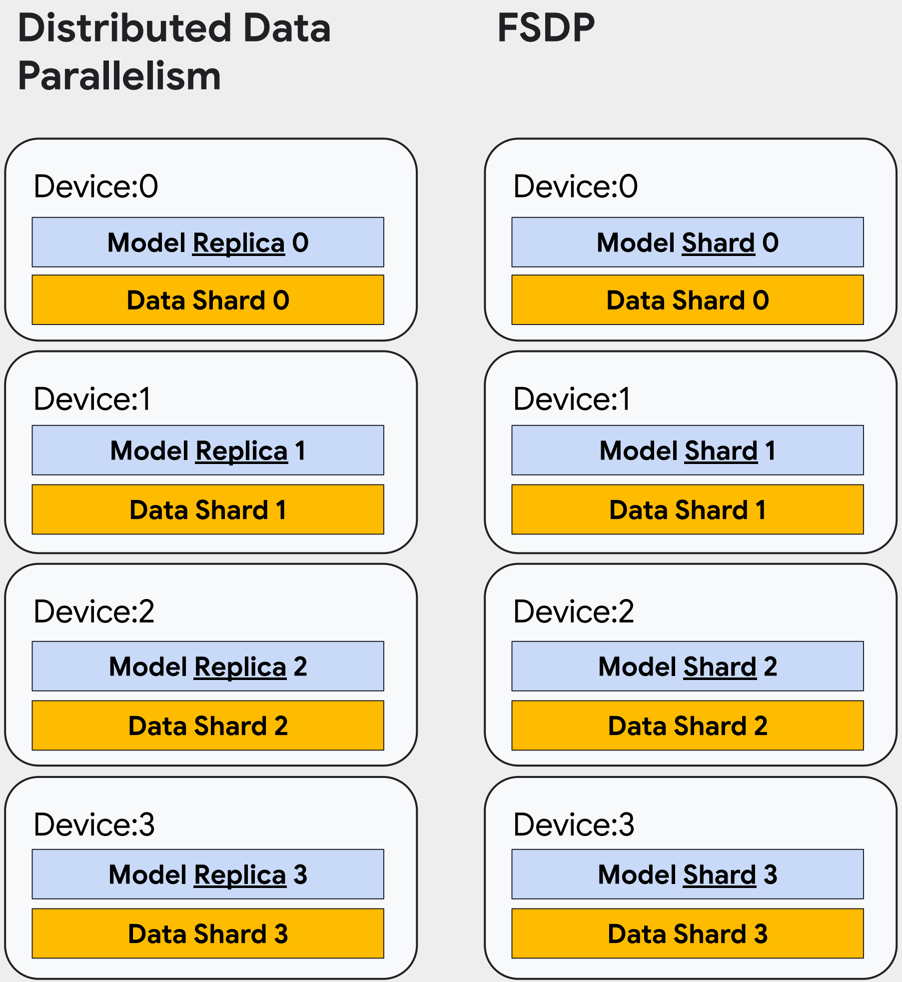
Distributed Data Parallelism

- **Model Replicated, Data Split**: Model copied to multiple devices (GPUs/TPUs); dataset divided into unique batches per device.
- **Parallel Gradient Calculation**: Each device computes gradients independently on its local data batch using its model copy.
- **Gradient Sync & Consistent Update**: Gradients aggregated across devices (e.g., averaged); combined result updates all model copies identically.



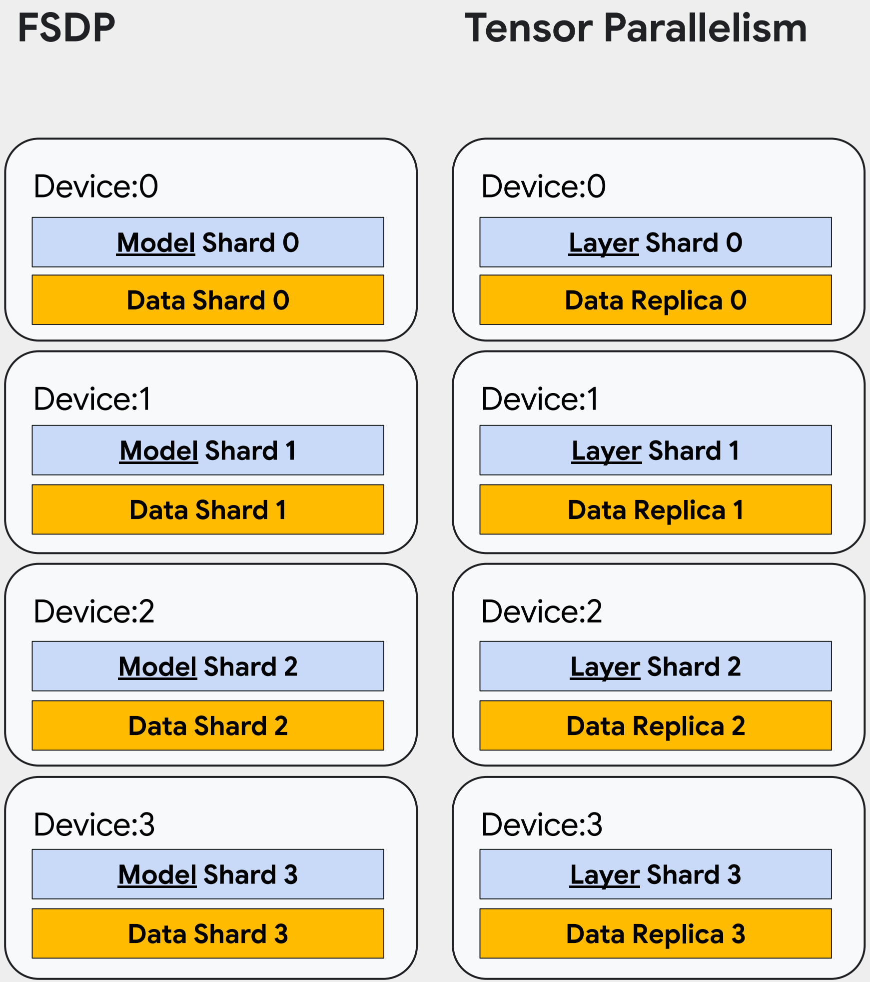
Fully Sharded Data Parallelism (FSDP)

- **Shards All State**: Partitions parameters, gradients, and optimizer states across devices
- **Cuts Memory Use**: Each device holds only its shard, greatly lowering memory needs
- **Gathers When Needed**: Assembles full layer parameters temporarily, just for computation



Tensor Parallelism

- **Splits Layers/Tensors**: A model parallelism technique that divides individual large model layers or tensor operations across multiple devices.
- **Cooperative Computation**: Devices work simultaneously on the same data input, each calculating only a portion or slice of the layer's computation.
- **Enables Huge Layers**: Allows models with layers too large to fit in a single device's memory to be executed by distributing the layer's workload.



JAX Parallelism Primitives: The **`Mesh`**

- **`jax.sharding.Mesh`**: Represents a logical grid mapped onto your physical accelerator devices (GPUs/TPUs).
- **Named Axes**: You assign names to the grid's dimensions (e.g., 'data', 'model').
- **Purpose**: Defines the hardware topology for sharding specifications.

JAX Parallelism Primitives: The **Mesh**

```python
# Example: 8 devices in a 4x2 grid
import jax
from jax.experimental import mesh_utils
from jax.sharding import Mesh

devices = mesh_utils.create_device_mesh((4, 2))
mesh = Mesh(devices, axis_names=('data', 'model'))

print(mesh)
# Output: Mesh(device_ids=array([[0, 1], [2, 3], [4, 5], [6, 7]]),
#             axis_names=('data', 'model'))
```

JAX Parallelism Primitives: `PartitionSpec`

- `jax.sharding.PartitionSpec` (or P): Describes how a tensor's dimensions map to `Mesh` axes.
- **Tuple Structure**: One element per tensor dimension.
- `'mesh_axis_name'`: Shard this dimension along the named mesh axis.
- `None`: Replicate this dimension across the named mesh axis.
- `P()`: Fully replicate the tensor on all devices in the mesh.

JAX Parallelism Primitives: `PartitionSpec`

```python
from jax.sharding import PartitionSpec as P

# On a ('data', 'model') mesh:
# Shard dim 0 on 'data', dim 1 on 'model'
spec1 = P('data', 'model')

# Shard dim 0 on 'data', replicate dim 1
# (Typical for input batches in data parallelism)
spec2 = P('data', None)

# Replicate dim 0, shard dim 1 on 'model'
# (Typical for weights in some model parallelism)
spec3 = P(None, 'model')
```

JAX Parallelism Primitives: `NamedSharding` & `device_put`

- **`jax.sharding.NamedSharding`**: Combines a **`Mesh`** and a **`PartitionSpec`** into a concrete, reusable sharding strategy.
- **`jax.device_put`**: Explicitly places data (e.g., NumPy arrays) onto devices with a specific Sharding. Essential for distributing input data.

JAX Parallelism Primitives: `NamedSharding` & `device_put`

```
from jax.sharding import NamedSharding, PartitionSpec as P
import numpy as np
import jax

# Assuming 'mesh' is the 4x2 ('data', 'model') mesh

# Sharding for input data (batch x features)
data_sharding = NamedSharding(mesh, P('data', None))

# Create some data and shard it
numpy_batch = np.arange(32 * 128).reshape((32, 128))
sharded_batch = jax.device_put(numpy_batch, data_sharding)

print(sharded_batch.sharding)
# Output: NamedSharding(mesh=..., spec=PartitionSpec('data', None))
```

Google

JAX Parallelism Primitives: `jax.jit` and Constraints

- `jax.jit`: JAX's Just-In-Time compiler. Triggers SPMD compilation when inputs are sharded.
    - "**Computation follows data**": Operations are partitioned based on input sharding.
    - Automatically inserts communication (e.g., all-reduce).
- `jax.lax.with_sharding_constraint`:
Inside a `@jax.jit` function, explicitly asserts or enforces a `PartitionSpec` on an intermediate value.
    - Guides the compiler, potentially inserting resharding operations if needed.

Flax NNX Quick Recap

- **Stateful Modules**: `nnx.Module` instances hold their state (parameters, batch stats) directly as attributes (`nnx.Param`, `nnx.BatchStats`). Closer to PyTorch's `nn.Module`.
- **Eager Initialization**: Parameters are typically created in `__init__`.
- **Metadata**: `nnx.Variable` types can hold arbitrary metadata. This is key for sharding!
- **Mutability vs. JAX**: NNX modules are mutable Python objects, but JAX transformations (`jit, grad`) require pure functions and immutable PyTrees.

Bridging NNX State and JAX Transformations

- **Problem**: How to use mutable NNX objects with JAX's functional `jit`, `grad`, etc.?
- **Solution 1: Functional API**:
  - `nnx.split(module) -> GraphDef (static), State (dynamic PyTree)`.
  - Pass State through JAX transforms (`jit`, `grad`).
  - `nnx.merge(graphdef, state)` or `nnx.update(module, state)` to reconstruct/update.
- **Solution 2: NNX Transformations**:
  - `nnx.jit`, `nnx.grad`, `nnx.vmap` handle splitting/merging automatically. More convenient.

Annotating Sharding in NNX: `flax.nnx.spmd`

- **Goal**: Embed sharding specifications (`PartitionSpec`) directly within the `nnx.Module` definition using metadata.
- `flax.nnx.spmd.with_partitioning` / `nnx.with_metadata`: Wrappers to attach sharding info during variable initialization.
- **Direct Annotation**: `nnx.Param(..., sharding=P(...))` often works too.
- **Result**: The `nnx.Variable` gets a `.sharding` attribute storing the `PartitionSpec` tuple. These are just hints for the compiler.

Annotating Sharding in NNX: `flax.nnx.spmd`

```python
# Inside an nnx.Module __init__
from flax import nnx

init_fn = nnx.initializers.lecun_normal()
rng_key = nnx.make_rng(0) # Example RNG key

# Using with_metadata (preferred)
self.kernel = nnx.Param(
  nnx.with_metadata(init_fn, sharding=(None, 'model'))(rng_key, shape)
)
# Or directly (if supported by Variable type)
self.bias = nnx.Param(
  nnx.initializers.zeros(rng_key, bias_shape), sharding=('model',)
)
```

Google

Workflow: The Sharded Initialization Function (1/3)

- **Problem**: Initializing a huge model directly might cause Out-Of-Memory (OOM) on the default device (e.g., device 0) before sharding.
- **Solution**: Use `@nnx.jit` (or `@jax.jit` with Functional API) to orchestrate initialization and apply sharding constraints within the compiled function.
- Steps Inside the Jitted Function:
  - 1. Instantiate the unsharded NNX module (still uses metadata).
  - 2. Extract the functional State PyTree: `state = nnx.state(model)`.

Workflow: The Sharded Initialization Function (2/3)

- Steps Inside the Jitted Function (cont.):
  - 3. Extract the PartitionSpec PyTree from metadata: `pspecs = nnx.spmd.get_partition_spec(state)`.
  - 4. Apply sharding constraints to the State:
    ```
    sharded_state = jax.lax.with_sharding_constraint(
                                            state, pspecs)
    ```
    .
    This tells the compiler the desired final layout.

Workflow: The Sharded Initialization Function (2/3)

```python
# Inside the @nnx.jit function (continued from previous)
# Assume 'model' and 'state' exist
from flax import nnx as nnx
import jax

# 3. Extract PartitionSpec PyTree from metadata
pspecs = nnx.spmd.get_partition_spec(state)

# 4. Apply constraints to the state PyTree
# This is where JAX/XLA plans the distribution
sharded_state = jax.lax.with_sharding_constraint(state, pspecs)
```

Workflow: The Sharded Initialization Function (3/3)

- Steps Inside the Jitted Function (cont.):
  - 5. Update the original module object with the now sharded state:

    ```
    nnx.update(model, sharded_state)
    ```

  - 6. Return the model.
- **Execution Context**: Call this jitted function within a `jax.sharding.Mesh` context to actually do the sharding.

Workflow: The Sharded Initialization Function (3/3)

```python
@nnx.jit # Decorate the whole initialization function
def create_sharded_model(model_args...):
  model = MyNNXModule(...) # Step 1
  state = nnx.state(model) # Step 2
  pspecs = nnx.spmd.get_partition_spec(state) # Step 3
  sharded_state = jax.lax.with_sharding_constraint(state, pspecs) # Step 4
  nnx.update(model, sharded_state) # Step 5
  return model # Step 6

# --- Execution ---
# Assume 'mesh' is defined
with mesh: # Step 7: Execute within the mesh context
  sharded_model = create_sharded_model(args...)
```

Advanced: Logical Axis Naming

- **Concept**: Annotate sharding using semantic names ('batch', 'embed', 'hidden') instead of physical mesh axes ('data', 'model').
- **`sharding_rules`**: A mapping (tuple of pairs) defining how logical axes map to physical mesh axes. E.g., ('batch', 'data'), ('hidden', 'model').
- **Usage**: Provide rules via `nnx.with_metadata(..., sharding_rules=...)` or attach later to `VariableState`.
- **Benefit**: Decouples model definition from specific hardware layout.

Using sharding_rules

```python
# The mapping from alias annotation to the device mesh.
sharding_rules = (('batch', 'data'), ('hidden', 'model'), ('embed', None))

class LogicalDotReluDot(nnx.Module):
  def __init__(self, depth: int, rngs: nnx.Rngs):
    init_fn = nnx.initializers.lecun_normal()

    # Initialize a sublayer `self.dot1`.
    self.dot1 = nnx.Linear(
      depth, depth,
      kernel_init=nnx.with_metadata(
        # Provide the sharding rules here.
        init_fn, sharding=('embed', 'hidden'), sharding_rules=sharding_rules),
      use_bias=False, rngs=rngs)
```

Building the Distributed Training Loop (1/2)

- **Shard Input Data**: Use `jax.device_put` with the appropriate `NamedSharding` (e.g., `P('data', None)`) for each batch before the training step.
- **Compile Training Step**: Wrap the main logic (forward, loss, grads, update) in a function decorated with `@nnx.jit`.
- **NNX State Management**: `nnx.jit` automatically handles passing the sharded model state in and propagating updates (parameters, optimizer state) back out.

Building the Distributed Training Loop (1/2)

```
# Inside training loop
import jax
from jax.sharding import NamedSharding, PartitionSpec as P

# Assume 'mesh' defined
input_sharding = NamedSharding(mesh, P('data', None))
numpy_batch, numpy_labels = get_next_batch() # Assume defined somewhere
sharded_batch = jax.device_put(numpy_batch, input_sharding)
# Assuming labels are 1D, shard batch dim
label_sharding = NamedSharding(mesh, P('data'))
sharded_labels = jax.device_put(numpy_labels, label_sharding)

# Call the compiled train_step
loss = train_step(sharded_model, optimizer, sharded_batch, sharded_labels)
```

Building the Distributed Training Loop (2/2)

- **Loss & Gradients**: Use `nnx.value_and_grad` (or `nnx.grad`). JAX AutoDiff works with sharded values, automatically inserting communication (e.g., gradient all-reduce).
- **Optimizer Updates**: Call `optimizer.update(grads)`. The `nnx.Optimizer` typically holds references to the sharded parameters and applies updates in a distributed manner. Optimizer state (like momentum) should also be sharded.

Building the Distributed Training Loop (2/2)

```
# Assume sharded_model, optimizer are NNX objects
@nnx.jit # Compile the entire step
def train_step(model, optimizer, batch, labels):
  def loss_fn(model_stateful): # loss_fn operates on the stateful model
    logits = model_stateful(batch) # Forward pass
    loss = jnp.mean(optax.softmax_cross_entropy_with_integer_labels(logits, labels))
    return loss

  # nnx.value_and_grad handles model state correctly
  loss_val, grads = nnx.value_and_grad(loss_fn)(model)

  # Optimizer updates model params (and its own state) in-place
  optimizer.update(model, grads)
  return loss_val
```

Data Loading with Grain

- **Need for Efficient Data Loading**: Essential for maximizing hardware utilization in distributed settings.
- **Grain**: Google's library for high-performance, deterministic data loading in JAX.
- **Built-in Sharding**: `grain.sharding.ShardByJaxProcess` automatically shards data based on `jax.process_index()` and `jax.process_count()`.
- **Integration**: Simplifies distributing data across multiple hosts and devices, working seamlessly with the JAX distributed setup.

Checkpointing Sharded Models

- **Challenge**: Saving/loading huge sharded models can cause OOM if gathered on one device.
- **Solution: Sharded Checkpointing:** Libraries like Orbax save/load individual tensor shards directly to/from devices.
- **NNX Metadata is Key**: Checkpointing needs the target sharding (**`NamedSharding`**) for each parameter to restore correctly.
- **`nnx.spmd.get_named_sharding`**: Utility to generate the required PyTree of **`NamedSharding`** objects from the model state and mesh, using the embedded **`.sharding`** metadata.

Checkpointing Sharded Models

```
# Assume 'sharded_model' and 'mesh' exist
# Assume 'checkpoint_mgr' is an Orbax CheckpointManager instance

# Get state structure (can use abstract state from nnx.eval_shape too)
state_struct = nnx.state(sharded_model) # Or nnx.state(abstract_model)

# Generate the target NamedSharding PyTree
target_shardings = nnx.spmd.get_named_sharding(state_struct, mesh)

# Use with Orbax (example)
checkpoint_mgr.save(ckpt_dir, args=orbax.args.StandardSave(sharded_model))
loaded_model = checkpoint_mgr.restore(checkpoint_mgr.latest_step(),
                 args=orbax.args.StandardRestore(target_shardings))
```

Key Considerations & Best Practices

- **Avoid Initialization OOM**: ALWAYS use the `create_sharded_model` pattern (initialize & constrain inside `@nnx.jit` within `Mesh` context).
- **Annotate Everything**: Ensure all relevant parameters have `.sharding` metadata.
- **Logical vs. Physical Axes**: Remember `with_sharding_constraint` needs physical mesh axis names, even if `params` use logical names.
- **Debugging**: Use `jax.debug.visualize_array_sharding` (or similar) to inspect layouts. Use constraints as assertions. Profile performance.
- **`nnx.jit` vs `jax.jit`:** `nnx.jit` is convenient; `jax.jit` + Functional API might offer slightly better performance (profile if needed).

Conclusion

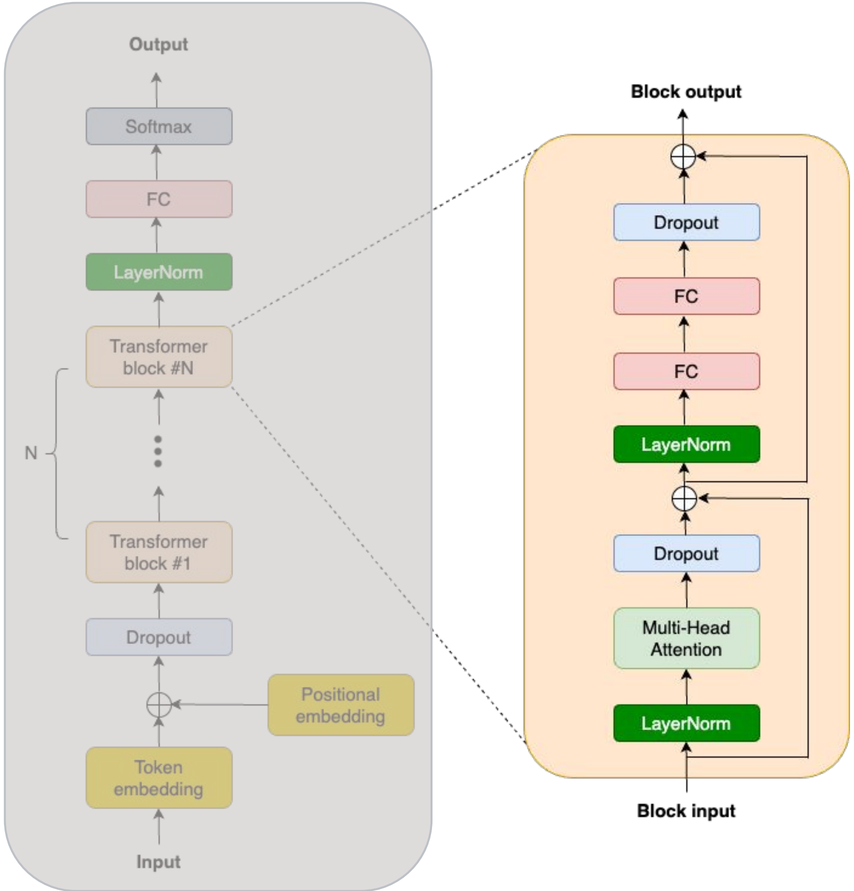
- JAX SPMD + Flax NNX provide a powerful way to scale large models.
- NNX's statefulness and metadata integrate naturally with JAX's explicit sharding primitives (`Mesh`, `PartitionSpec`).
- **Key Workflow**: Annotate metadata -> Initialize sharded via `@nnx.jit` + `with_sharding_constraint` -> Shard inputs -> Train with `@nnx.jit`.
- Enables building and training massive models while keeping core training logic relatively clean.

Sharding and parallelism in practice

A transformer block example with NNX

Google

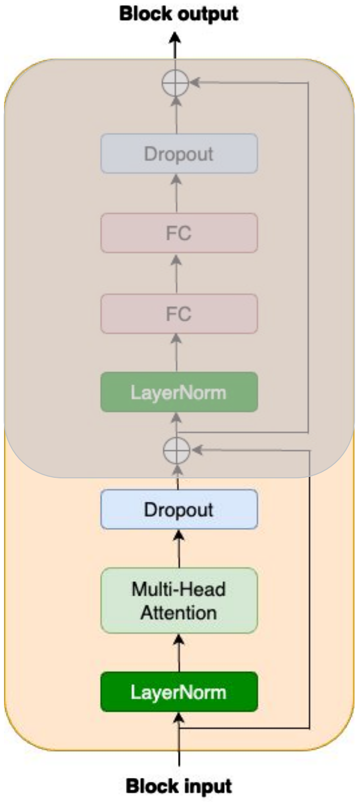
GPT2 architecture



Google

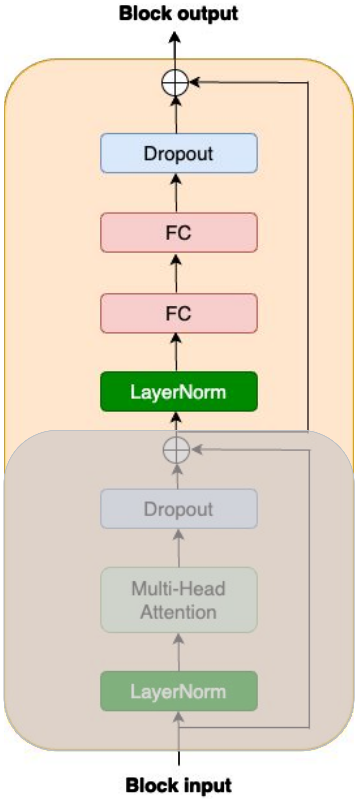
Transformer block

```python
class TransformerBlock(nnx.Module):
    def __init__(self, embed_dim: int, num_heads: int, ff_dim: int,
                 dropout_rate: float, rngs: nnx.Rngs):
        self.layer_norm1 = nnx.LayerNorm(
                                epsilon=1e-6,
                                num_features=embed_dim,
                                rngs=rngs)
        self.mha = nnx.MultiHeadAttention(
                        num_heads=num_heads,
                        in_features=embed_dim,
                        rngs=rngs)
        self.dropout1 = nnx.Dropout(rate=dropout_rate)
    ......
```



Transformer block (cont'd)

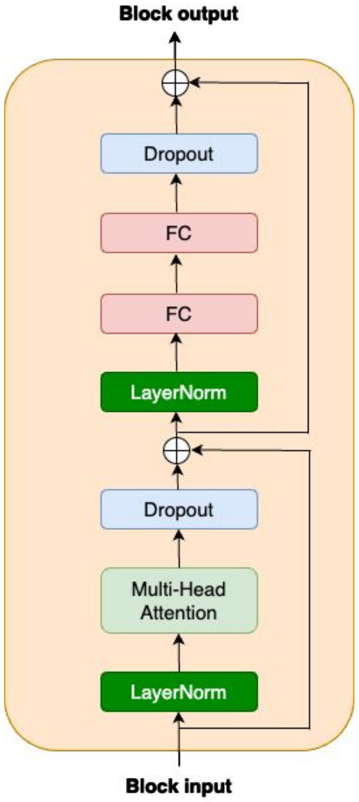
```python
class TransformerBlock(nnx.Module):
    def __init__(self, embed_dim: int, num_heads: int, ff_dim: int,
                 dropout_rate: float, rngs: nnx.Rngs):
        ......
        self.layer_norm2 = nnx.LayerNorm(epsilon=1e-6,
                                         num_features=embed_dim,
                                         rngs=rngs)
        self.linear1 = nnx.Linear(in_features=embed_dim,
                                  out_features=ff_dim,
                                  rngs=rngs)
        self.linear2 = nnx.Linear(in_features=ff_dim,
                                  out_features=embed_dim,
                                  rngs=rngs)
        self.dropout2 = nnx.Dropout(rate=dropout_rate)
```



Google

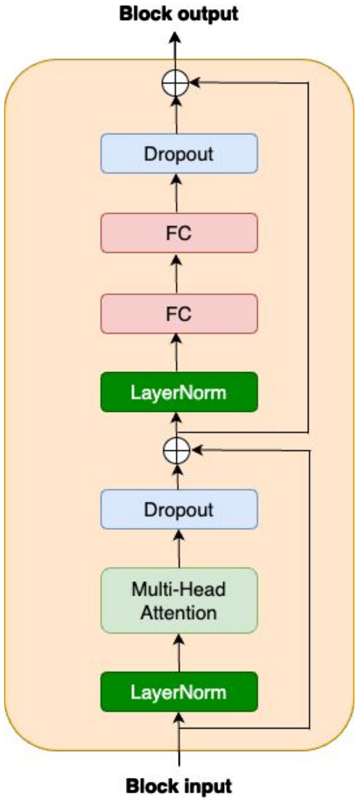
Transformer block (cont'd)

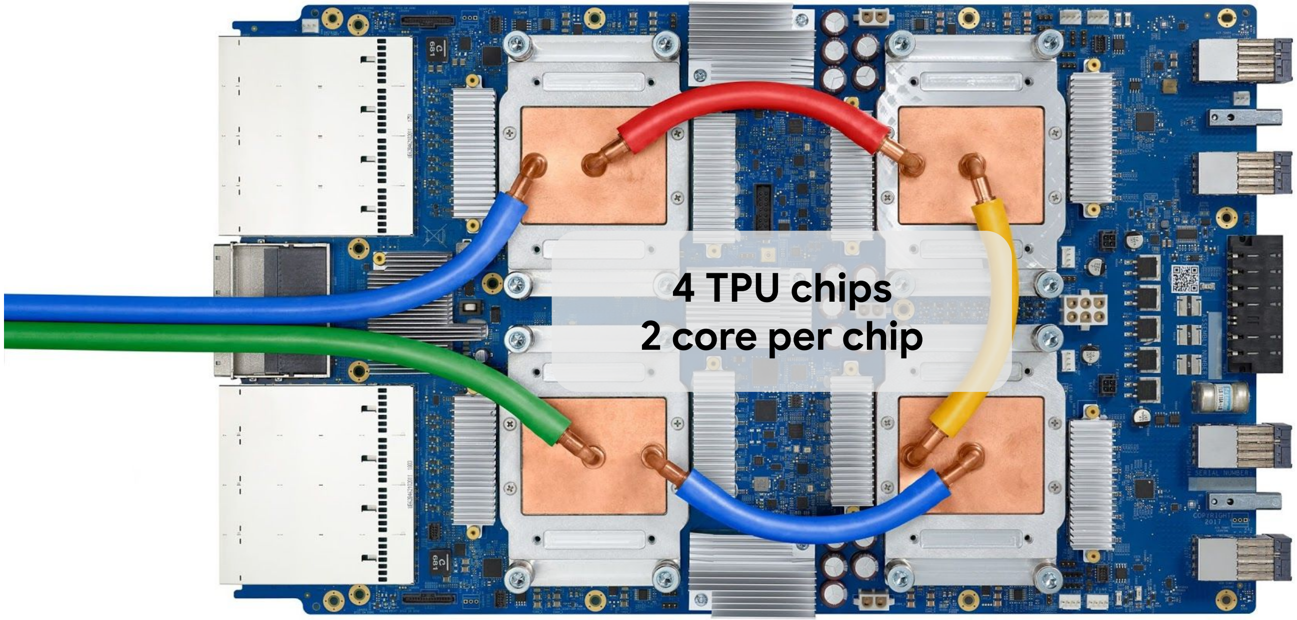
```
class TransformerBlock(nnx.Module):
    def __call__(self, inputs, training: bool = False):
        input_shape = inputs.shape
        bs, seq_len, emb_sz = input_shape

        attention_output = self.mha(
            inputs_q=self.layer_norm1(inputs),
            mask=causal_attention_mask(seq_len), decode=False,
        )
        x = inputs + self.dropout1(attention_output,
                                   deterministic=not training)
(… to be continued)
```



Transformer block (cont'd)

```
# MLP
mlp_output = self.linear1(self.layer_norm2(x))
mlp_output = nnx.gelu(mlp_output)
mlp_output = self.linear2(mlp_output)
mlp_output = self.dropout2(mlp_output, deterministic=not training)

return x + mlp_output
```

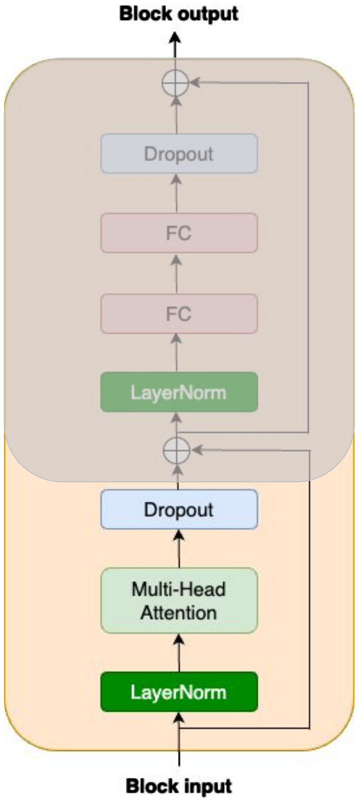




TPU v3

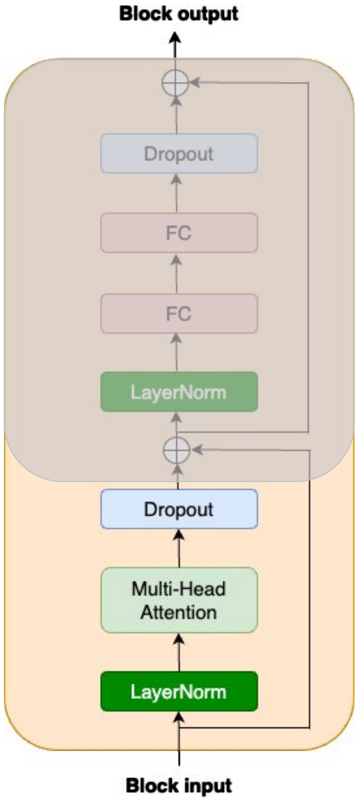
Transformer block sharding

```
mesh = Mesh(mesh_utils.create_device_mesh((4, 2)),
            ('batch', 'model'))

class TransformerBlock(nnx.Module):
    def __init__(self, embed_dim: int, num_heads: int, ff_dim: int,
                 dropout_rate: float, rngs: nnx.Rngs):
        self.layer_norm1 = nnx.LayerNorm(epsilon=1e-6,
                                          num_features=embed_dim,
                                          scale_init=nnx.with_metadata(
                                              nnx.initializers.ones_init(),
                                              sharding=('model',)),
                                          bias_init=nnx.with_metadata(
                                              nnx.initializers.zeros_init(),
                                              sharding=('model',)),
                                              rngs=rngs)
```



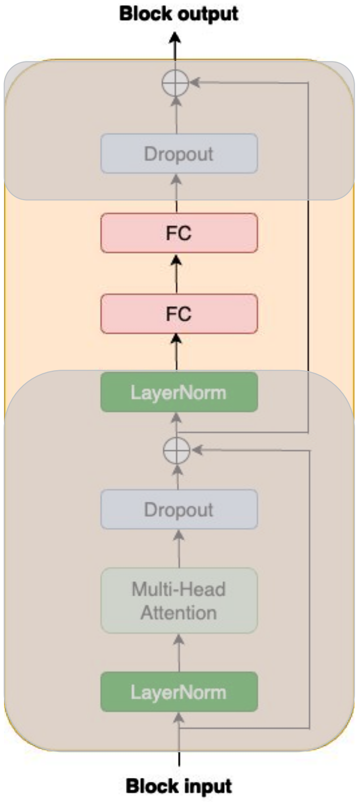
Transformer block sharding

```
self.mha = nnx.MultiHeadAttention(
              num_heads=num_heads,
              in_features=embed_dim,
              kernel_init=nnx.with_metadata(
                         nnx.initializers.xavier_uniform(),
                         sharding=('model',)),
              bias_init=nnx.with_metadata(
                        nnx.initializers.zeros_init(),
                        sharding=('model',)),
              rngs=rngs)
self.dropout1 = nnx.Dropout(rate=dropout_rate)
```



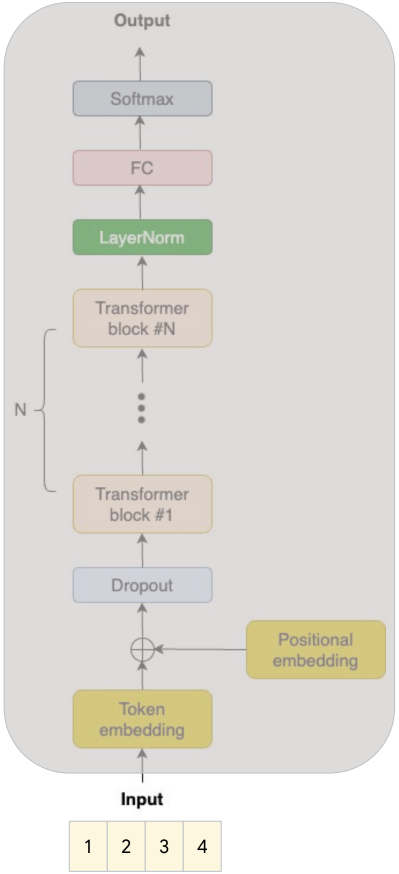
Transformer block sharding (continued)

```
self.linear1 = nnx.Linear(in_features=embed_dim,
                  out_features=ff_dim,
                  kernel_init=nnx.with_metadata(
                    nnx.initializers.xavier_uniform(),
                    sharding=('model',)),
                  bias_init=nnx.with_partitioning(
                    nnx.initializers.zeros_init(),
                    sharding=('model',)), …)
self.linear2 = nnx.Linear(in_features=ff_dim,
               out_features=embed_dim,
               kernel_init=nnx.with_metadata(
                 nnx.initializers.xavier_uniform(),
                 sharding=('model',)),
               bias_init=nnx.with_metadata(
                 nnx.initializers.zeros_init(),
                 sharding=('model',)), …)
```



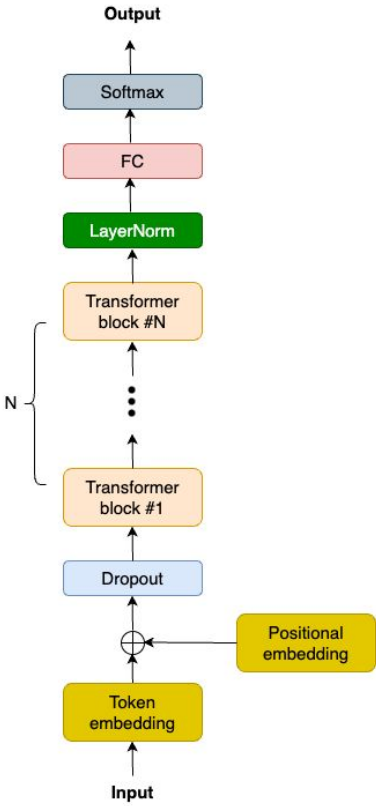
Data parallelism in the training loop

```
while True:
    input_batch, target_batch = get_batch("train")
    train_step(model, optimizer, train_metrics,
               jax.device_put((input_batch, target_batch),
                     NamedSharding(mesh, P('batch', None))))
```



Switching among different parallelisms (8 devices)

```
# 4-way batch data parallelism and 2-way model parallelism
mesh = Mesh(mesh_utils.create_device_mesh((4, 2)), ('batch', 'model'))

# 2-way batch data parallelism and 4-way model parallelism
mesh = Mesh(mesh_utils.create_device_mesh((2, 4)), ('batch', 'model'))

# Pure model parallelism
mesh = Mesh(mesh_utils.create_device_mesh((1, 8)), ('batch', 'model'))

# Pure batch data parallelism
Mesh = Mesh(mesh_utils.create_device_mesh((8, 1)), ('batch', 'model'))
```



Learning Resources

Code Exercises, Quick References, and Slides

- https://goo.gle/learning-jax

Learn JAX

Community and Docs

Community:

- https://goo.gle/jax-community

Docs

- JAX AI Stack: https://jaxstack.ai
- JAX: https://jax.dev
- Flax NNX: https://flax.readthedocs.io