GSPMD: generalized parallelism for large models as shared compiler infrastructure

Yuanzhong Xu

Google

Google Research

Model scaling: recent advances in several domains

Language: 100B ~ over 1T parameters

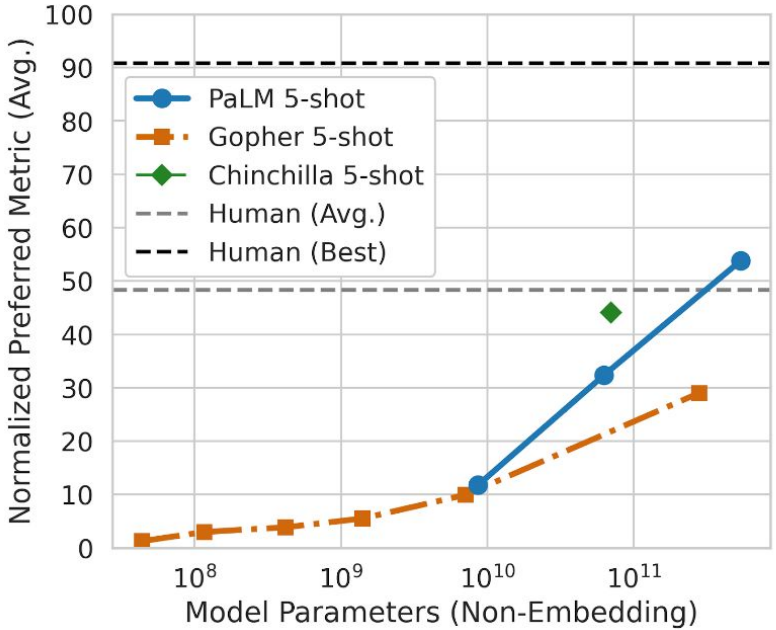
- GPT-3
- M4 Translate
- GLaM
- PaLM

Vision + text: several billion to 20B parameters:

- DALLE 2
- Imagen
- Parti
- Stable diffusion, …

Speech: 10B parameters

- BigSSL: Google



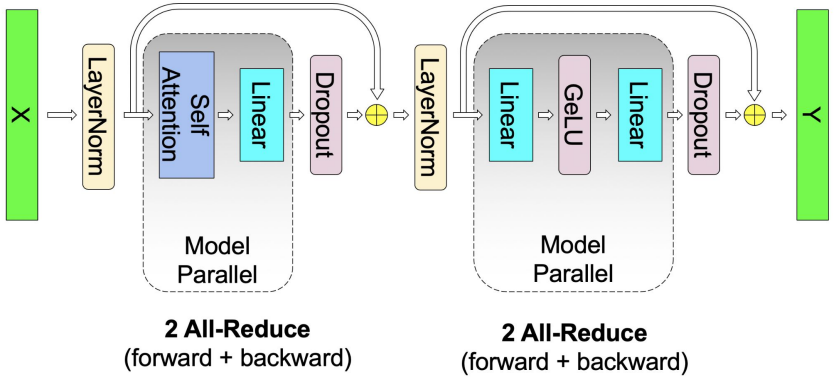
Large models: model-specific communications

For comparison: small models + large data

- Data parallelism: all communications are done on the gradients

Large models

- Communication within the forward and backward passes. Specific to the maths of a layer.



E.g., Megatron LM model parallelism in transformer layers.

Desired properties for a shared infrastructure

- Separation of concern
  - Separate partitioning and communication from the maths of a layer
- Easy reconfiguration
  - A model implementation can be configured with different model sizes, device cluster topologies, training vs serving modes.
  - Partitioning and communication can be very different, but ideally they can be achieved by simple reconfigurations.
- Reusability
  - The same core infrastructure can be shared by different models, frameworks, and hardware platforms
- Performance
  - Common optimizations can be implemented in this infrastructure.

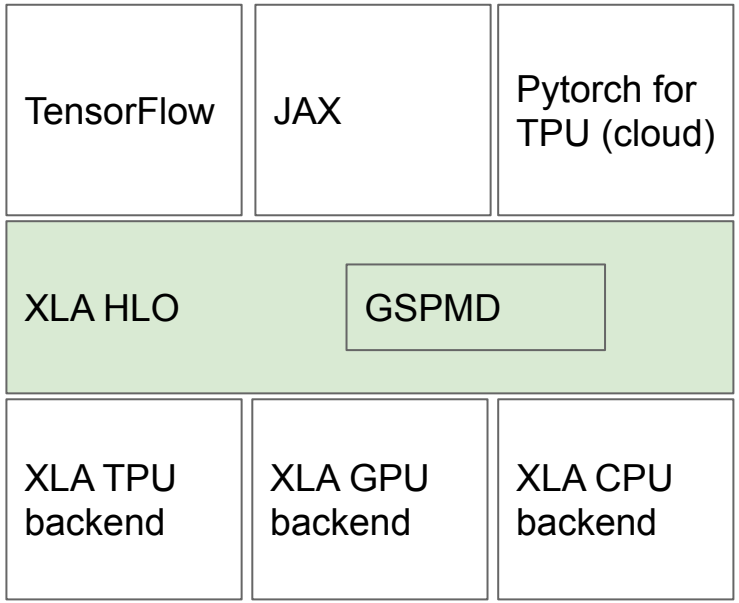
GSPMD: compiler-based shared infrastructure

XLA HLO: shared IR for multiple frontends and hardware platforms

- Small yet expressive op set

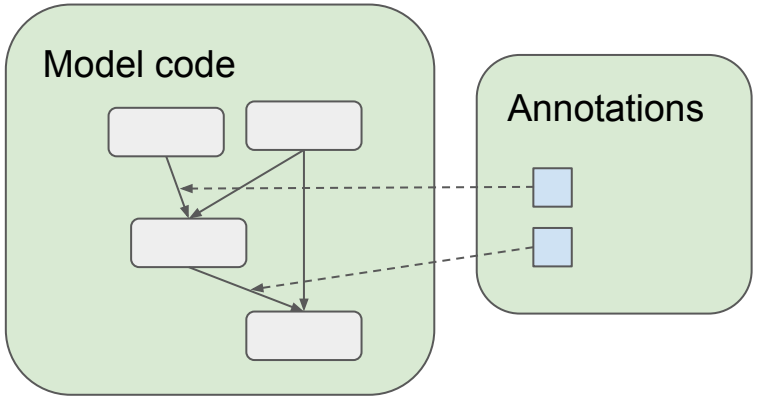
GSPMD

- Built at the level of XLA HLO

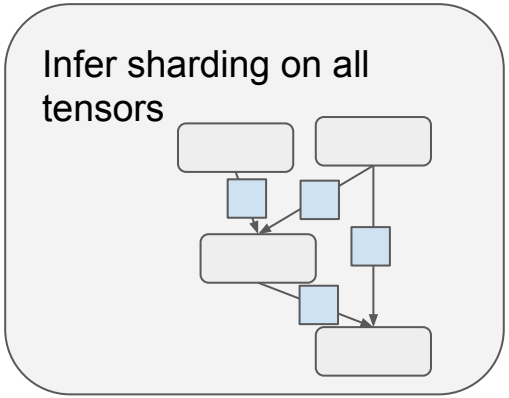


GSPMD approach: decoupling model and parallelism

Write model code as if there were a single, large device

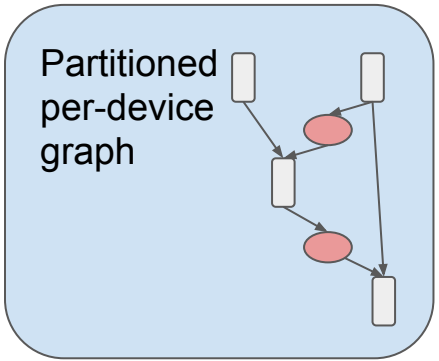


Annotate some key tensors for sharding



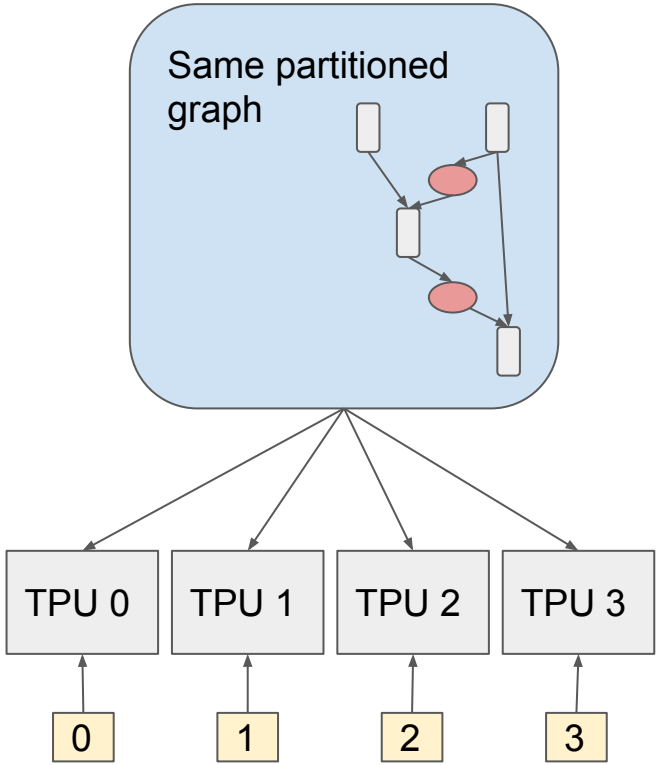
XLA SPMD partitioner

Partitioned graph with collective cross-core communication ops



Single Program, Multiple Data (SPMD) Partitioning

- One-time compilation for all partitions, fast compilation to thousands of partitions
- Avoids cross-program scheduling problems
- Same program runs on all partitions
  - Runtime sets partition IDs
  - Program calculates offsets, padding, etc based on runtime partition ID



Adoptions

Google internal

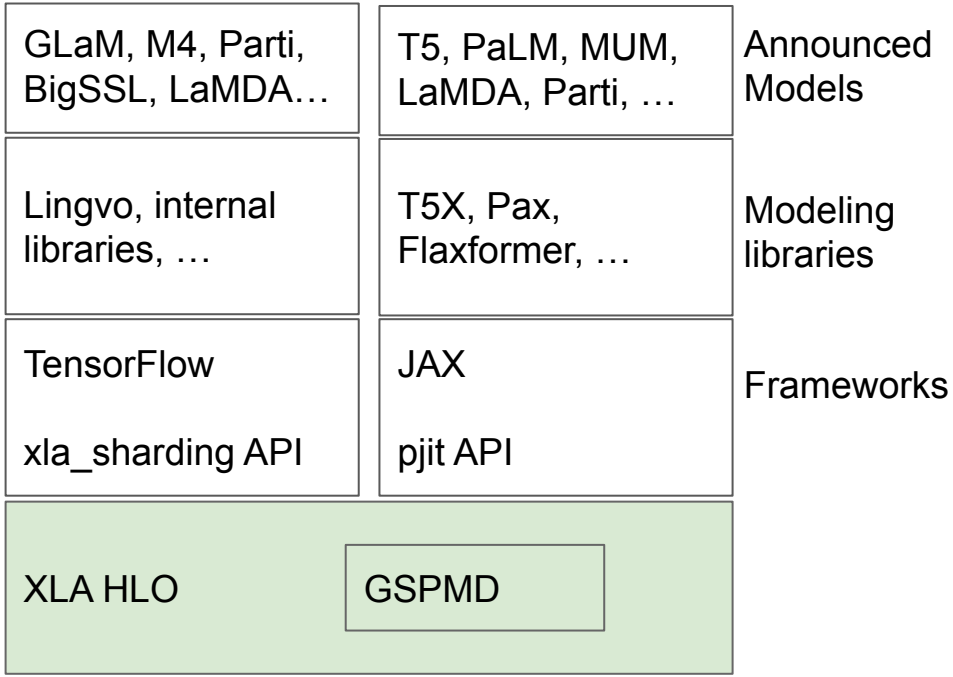
- Adoptions across the stack
- Enabled research in multiple domains: language, speech, vision, multimodal, …
- Various production-focused projects

Cloud TPUs

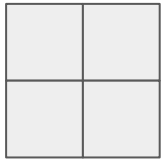
- Cohere AI
- LG AI Research
- …

MLPerf:

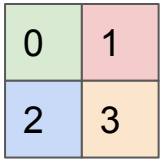
- Large model training
- Small model performance scaling



Low-level sharding abstraction: per-tensor annotation

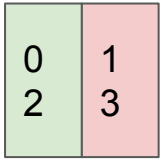


2D tensor on 4 partitions



2. Tiled:
- Every partition has one ¼ data
- Device order can be specified

1. Replicated:
- Every partition has the full data



3. Partially Tiled:
- Replicated in subgroups
- Each subgroup has a different subset of data

Describes only how data is distributed, not how sharded ops are implemented.

Per-tensor annotation: enables switching between parallelism modes

Framework-level APIs

Framework usually use a higher-level abstraction

- Mesh and axes: sharding is specified as a mapping from mesh axes to tensor dimensions
- Device mesh typically stays constant in the whole program. Axes mappings change from tensor to tensor

Examples:

- TensorFlow mesh_split()
- JAX pjit()

IR coverage and advanced cases

- XLA HLO has a small op set so we can cover the entire IR
- Improved coverage over the past 2 years

Advanced features

- Uneven partitioning: automatically pads data and masks invalid areas
- Convolution halo exchanges: spatial partitioning of images
- Multi-dimensional partitioning: recursive pattern match
- Coverage of irregular ops: e.g., slice, concatenate, reshape, …
    - These are less noticeable from the high-level model maths, but they are tricky to handle and often cause problems if not handled efficiently.

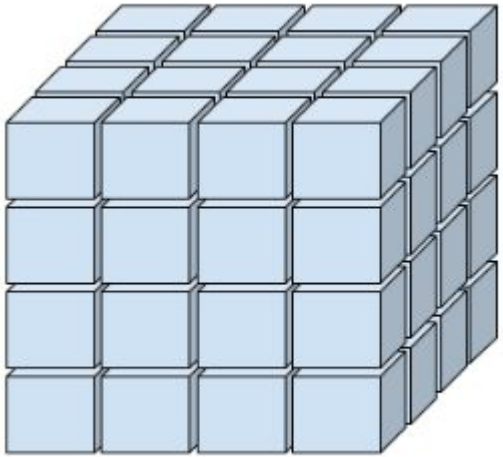
Sharding propagation

- Reduces the amount of required user annotations
  - Annotating a few tensors in Transformer can achieve all common types of parallelism and combinations of them.
- We use a priority-based algorithm to iteratively refine shardings on the graph
  - Merge compatible/orthogonal shardings
  - Propagate in both forward and backward directions
  - Work through control flow (loops, conditionals)
  - Use a priority to make the decision more intuitive: e.g., elementwise ops have highest priority so they typically don’t change shardings.

Large-scale sharding on TPU pods

- 2D mesh/torus on TPUv2, v3
- 3D torus on TPUv4
- Thousands of interconnected devices, high bandwidth

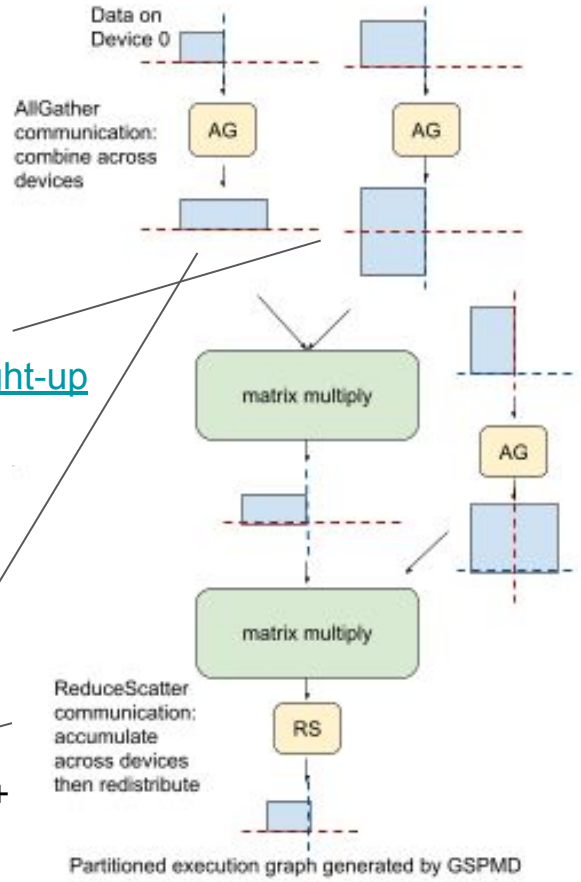
- In-layer sharding can be applied at a very large scale



Transformer training on a TPUv4 pod

Fully-partitioned 2D parallelism

- A simple configuration that incorporates several techniques
- Achieves 63% FLOPS utilization on 200B~480B models on 2048 TPU chips
- No need for activation recompute in the backward pass.

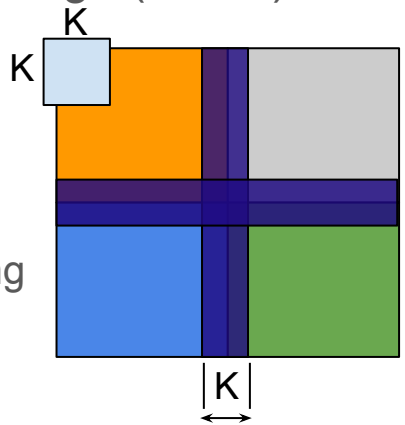


Fully-sharded data parallelism/Zero/weight-up date sharding

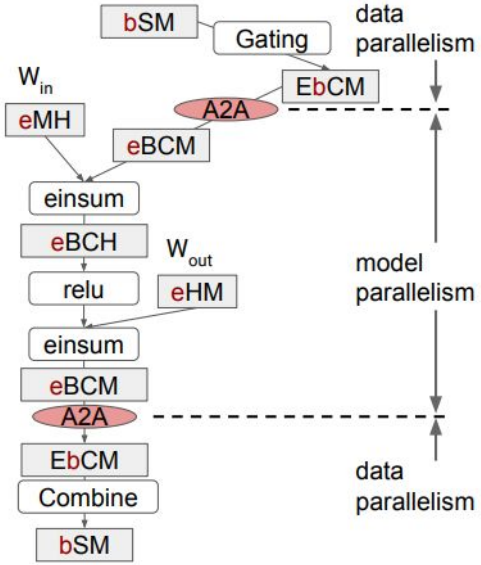
Megatron model-parallelism + activation scatter/gather

Easy configurations on more types of models

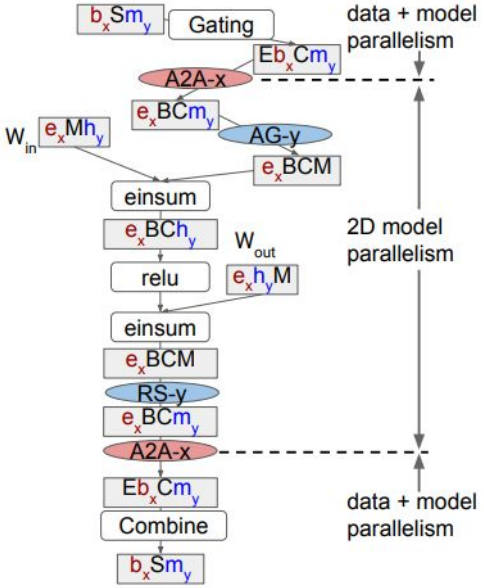
- Mixture-of-experts (right) in GLaM: 1D and 2D shardings
- Other types of models: e.g., image spatial partitioning with automatic halo exchange (below).



Spatial partitioning



1D MoE

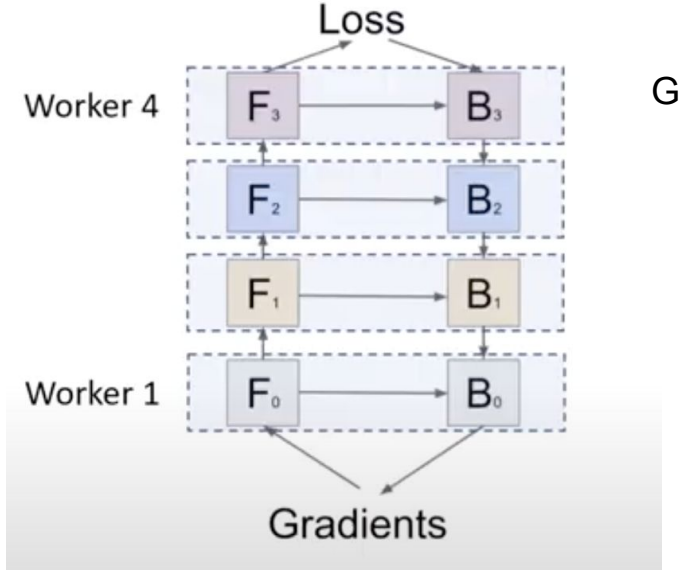


2D MoE + in-expert sharding

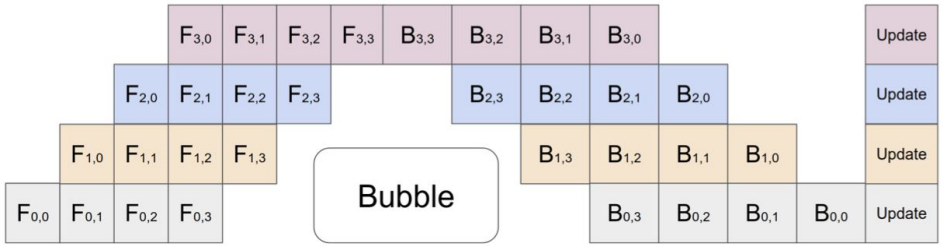
Pipeline parallelism

Pipeline parallelism: cross-layer graph partitioning, breaks data into “microbatches”. It has very small communication overhead, but has 2 challenges

- Pipeline “bubbles”: idle time due to data dependency
  - Could be optimized with advanced pipeline schedules
- Recompute in the backward pass
  - In the simplest implementation, the whole fprop is recomputed in bprop to avoid complexity around loop structures.
  - Also a solvable problem



GPipe



Why we need pipelining at Google

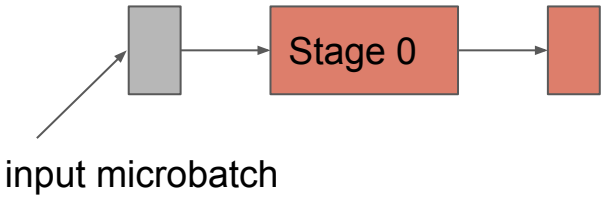
With large TPU pods, pipelining is less critical compared to GPU systems. However, we still need it for

- Flexibility on using multiple TPU pods (or subsets of pods). Sometimes it’s easier to schedule and/or optimize on smaller connected topologies.
- Performance. Even within a TPU pod, pipeline can be better than in-layer sharding for “deep-and-thin” type of Transformer configurations.

Pipeline with GSPMD

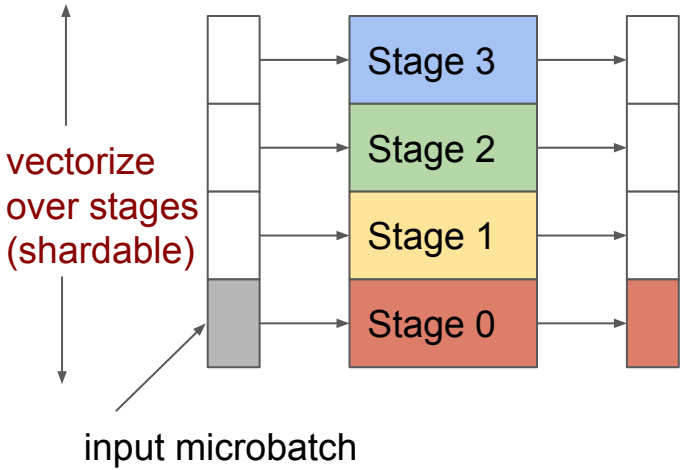
- As an in-layer partitioning system, GSPMD is compatible with/orthogonal to additional pipelining infrastructure
  - GSPMD can be used to partition each pipeline stage
- On the other hand, we introduce a new method, the GSPMD pipeline approach
  - It has been successful in many important use cases, including the publicly known BigSSL speech model, and the Parti text-to-image model

Alternatively, pipeline is a recurrent layer…

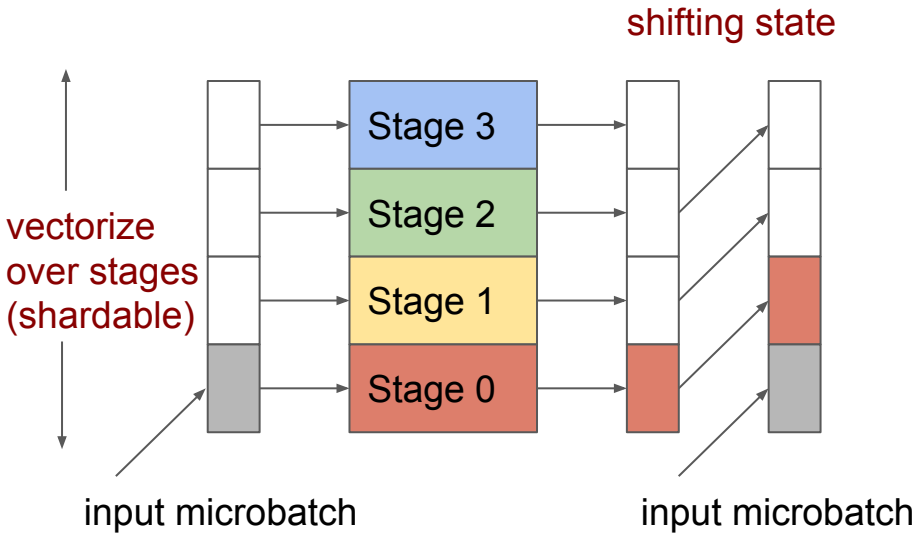


Alternatively, pipeline is a recurrent layer…

with vectorization…



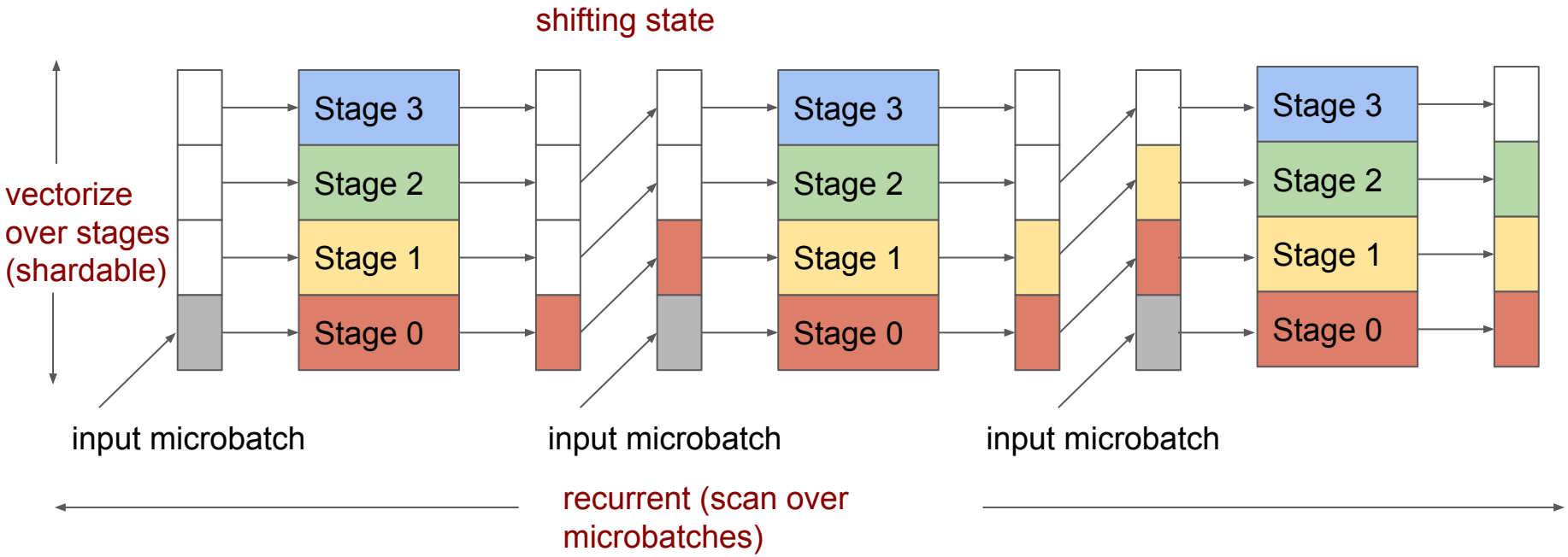
Alternatively, pipeline is a recurrent layer…



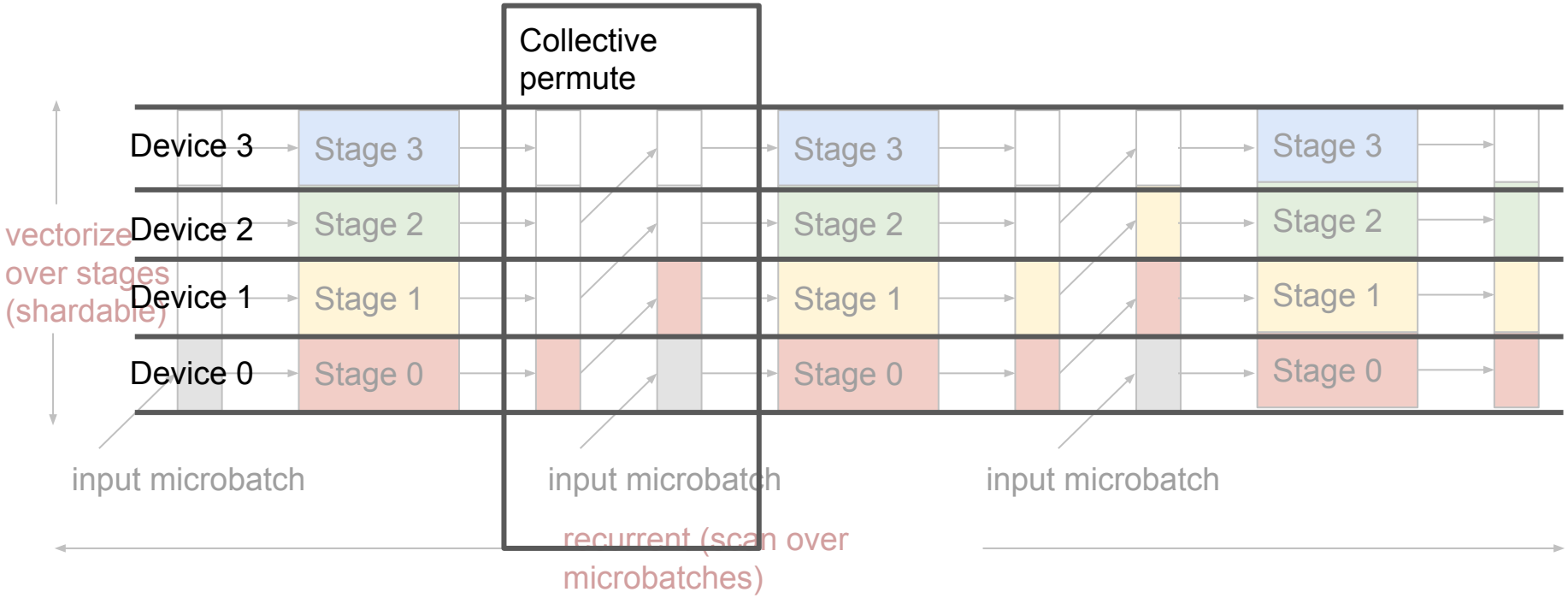
with vectorization…

and a shifting state…

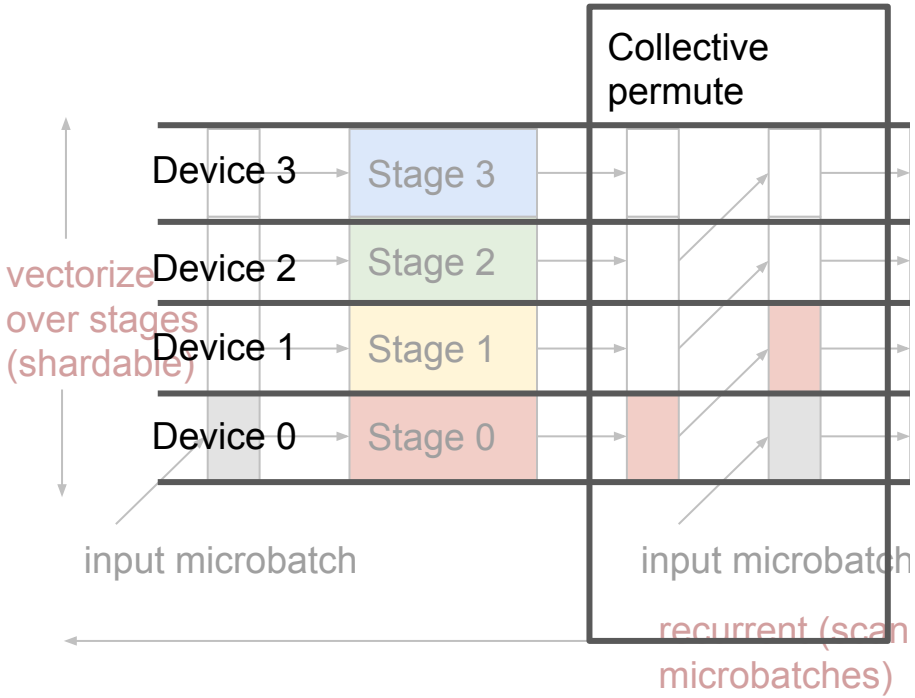
Alternatively, pipeline is a recurrent layer…



GSPMD pipeline: pipeline as sharding



GSPMD pipeline: communication performance



- An optimized implementation can make the data transfer completely overlap with stage compute
- Depending on the backend and mesh configurations, transfers can be done on either fast interconnects or host networks

GSPMD pipeline: limitations and advantages

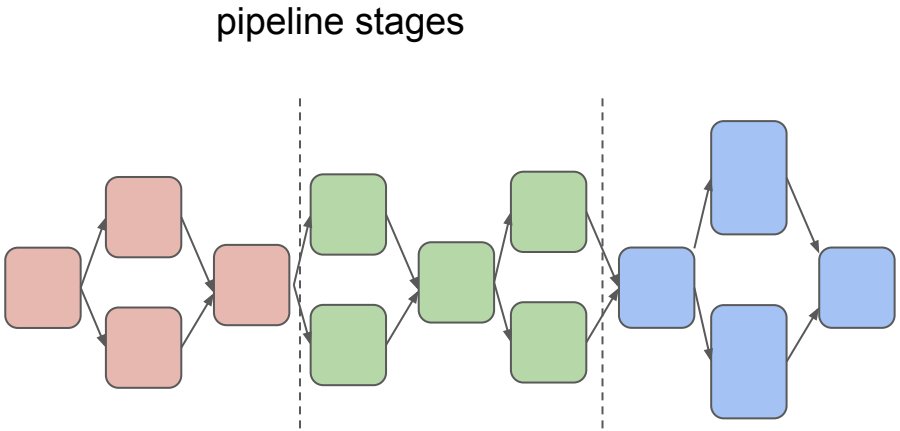
Obvious limitation: supports only homogeneous pipeline stages. However, it is more useful than one might think

- Transformer-like models are all supported, and they are the majority of recent large models
- Encoder-decoder models are also supported

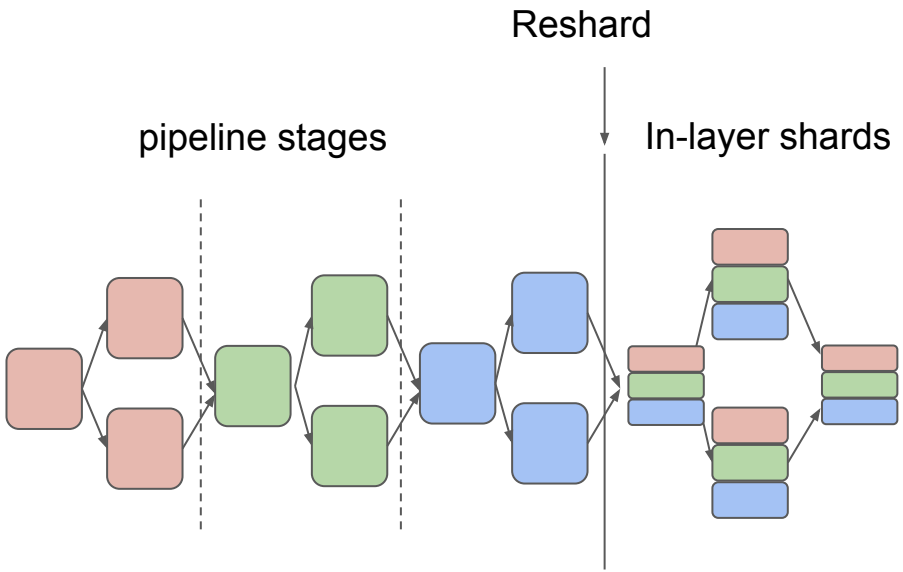
Advantages:

- Modularity. Pipeline parallelism can be easily applied to part of the model.
- Reuses existing optimizations for single-core programs
  - JAX selected recompute (“rematerialization”)
  - XLA optimizations on loops
- Simplicity. Does not require new capabilities from low-level systems.
- Trivial to support weights shared across stages. E.g., shared softmax and embedding weights.

GSPMD pipeline: modularity



Typical pipeline implementation:
sequentially divide the whole graph



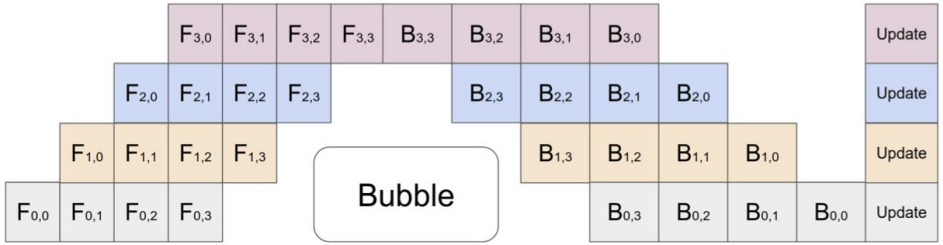
GSPMD pipeline is a local configuration on a subgraph

GSPMD pipeline implementation and usage

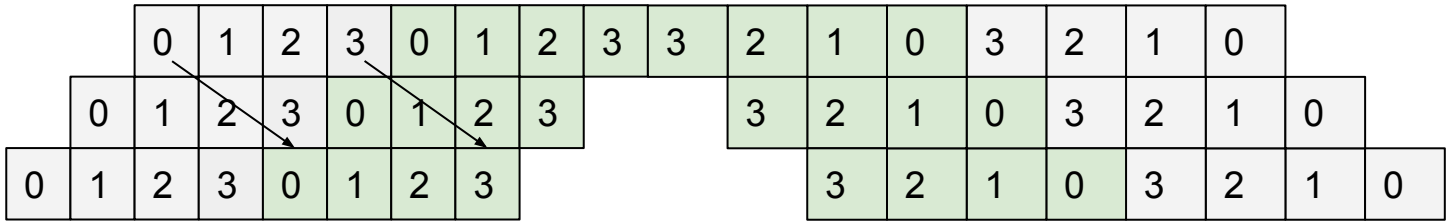
A wrapper layer in a modular model library. Implemented in two public libraries: Lingvo (TensorFlow) and PAX/Praxis (JAX). They each include 2 pipeline schedules optimized for small- and large-batch cases. Example usage:

```
# Stage definition
one_stage_params = TransformerLayer.HParams(…)

# Set up pipeline
pipelined_layer_params = pipeline.LayerwiseShardablePipelined.HParams(
          name='pipeline',
          num_stages=4,
          single_stage_body=one_stage_params)

# Sharding on the stage dimension
pipelined_layer_params.weight_sharding.stages = ['stages']
```

Implemented pipeline schedules



GPipe. Large bubble ratio, good for large-batch cases.



Forward Backward

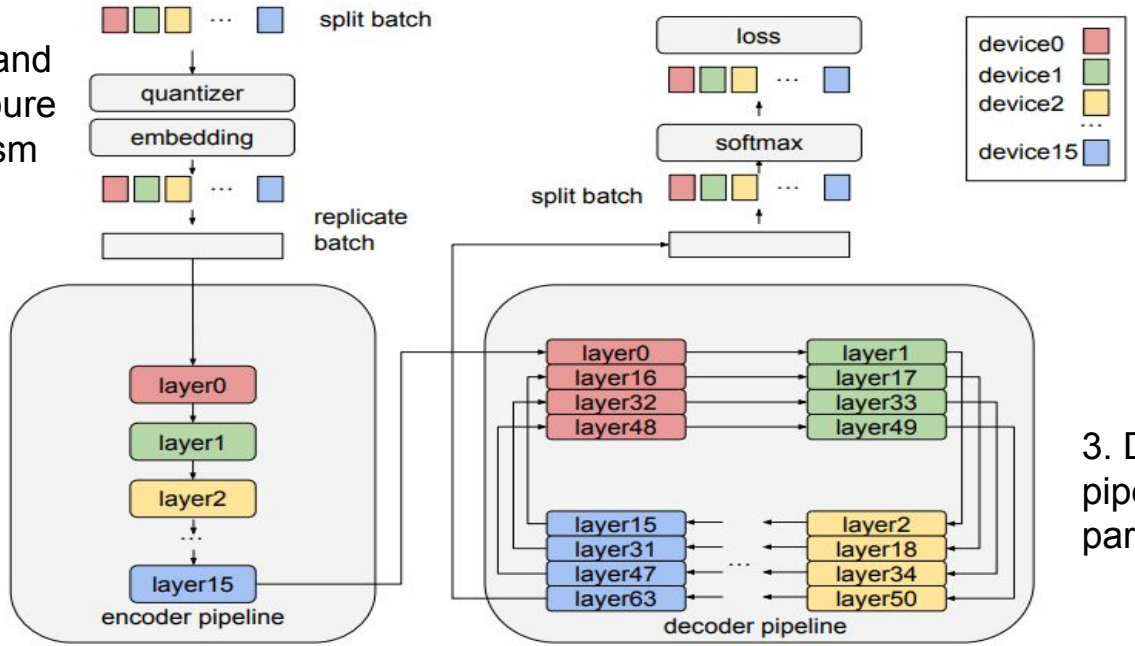
Circular. Smaller bubble ratio, good for small-batch cases.

- Interleaved layer assignment. 4-stage, 2-way circular schedule shown above. Similar ideas have been used for GPUs outside Google (e.g., new Megatron paper in 2021)

Parti use case: modular pipeline + sharding

Parti is a text-to-image model using Transformer encoder-decoder architecture.

1. Quantizer and embedding: pure data parallelism



4. Softmax and loss: pure data parallelism

2. Encoder: pipeline (+ data parallelism not shown)

3. Decoder: circular pipeline (+ data parallelism not shown)

Parti use case: modular pipeline + sharding

- Modularity
  - pipeline parallelism is local to a subcomponent
- Performance
  - perfect load balancing across stages by not including other layers into the pipeline
  - Circular pipeline schedule dramatically reduces the bubbles
- Flexibility
  - switching between data- and pipeline-parallelism

GSPMD takeaways

- Parallelism is a layer-local property. Even pipelining can be made modular.
- Avoid over-specialization: a group of techniques are achieved as merely different configurations.
- Non-intrusive APIs accelerates adoption. Parallelism is separated from user code to configurations and low-level optimizations.

- A good foundation is important: XLA HLO.
- Details matter as much as high-level designs: need to solve tricky problems like how to partition a convolution and a slice op.