SCALING DEEP LEARNING TRAINING WITH MPMD PIPELINE PARALLELISM

Anxhelo Xhebraj [1] **Sean Lee** [1] **Hanfeng Chen** [1] **Vinod Grover** [1]

ABSTRACT

We present JaxPP, a system for efficiently scaling the training of large deep learning models with flexible pipeline parallelism. We introduce a seamless programming model that allows implementing user-defined pipeline schedules for gradient accumulation. JaxPP automatically distributes tasks, corresponding to pipeline stages, over a cluster of nodes and automatically infers the communication among them. We implement a MPMD runtime for asynchronous execution of SPMD tasks. The pipeline parallelism implementation of JaxPP improves hardware utilization by up to $1.16\times$ with respect to the best performing SPMD configuration.

1 INTRODUCTION

The capability of deep learning models in a wide array of tasks has been shown to scale with model size (Brown et al., 2020; Dosovitskiy et al., 2021). Consequently, researchers are training increasingly larger models (Shoeybi et al., 2020; Chowdhery et al., 2022). Considerable development efforts are required to run such experiments, which are often justified only for well-performing models, thus restricting the exploration of models that excel when scaled (Hooker, 2020).

A primary challenge in developing large models lies in their efficient parallelization across various hierarchies (cores, devices, hosts, data centers) to maximize resource utilization and minimize device communications and tensor layout changes (Shoeybi et al., 2020; Narayanan et al., 2021; Fedus et al., 2022; Pope et al., 2022). Early works concentrated on manually re-implementing models to run them across multiple devices, which resulted in highly optimized runtimes for specific models (Shoeybi et al., 2020; Narayanan et al., 2021). However, this approach is time-consuming and error-prone.

The GSPMD programming model, as implemented in the XLA compiler (Xu et al., 2021), simplifies the parallelization of linear algebra workloads. It requires only lightweight annotations that specify how tensors in a computation should be sharded across a mesh of devices, with the compiler automatically handling the placement of collective operations for communication. Once the tensors are sharded and collective operations are in place, the computation is carried out in a Single-Program Multiple-Data (SPMD) fashion. This decoupling of sharding annotations from computation definitions facilitates experimentation with various intra-operator parallelism strategies (Zheng et al., 2022) such as data parallelism and tensor parallelism to minimize latency (Fedus et al., 2022; Pope et al., 2022).

Despite GSPMD's near-ideal solution, the SPMD model works well in practice only when high-bandwidth links connect accelerators, e.g., NVSwitch for GPUs and ICI for TPUs. This is because the collective operations necessary for the SPMD computations stress the network's bandwidth. It is well-known that scaling high-bandwidth links to larger device meshes quickly becomes infeasible. For example scaling the training of LLMs on TPUs (Chowdhery et al., 2022) required designing a separate system (Barham et al., 2022) to extend the SPMD model to cross low-bandwidth (DCN) domains. In settings where device connection has a low bandwidth, communication overhead can be greatly reduced with pipeline parallelism (Huang et al., 2019), which requires only Point-to-Point (P2P) communication.

GSPMD can implement only one variant of pipeline parallelism, precluding any form of pipeline parallelism that requires a Multiple-Program Multiple-Data (MPMD) paradigm. This limitation restricts significantly the types of computations that can be pipelined, and precludes various pipeline schedules that improve throughput and memory usage. In practice, best performing training configurations (Shoeybi et al., 2020) use a mix of pipeline, tensor, and data parallelism. Tensor parallelism is mapped over the high-bandwidth mesh dimension while data and pipeline parallelism are mapped over the low-bandwidth dimension.

Our work introduces JaxPP, a system for distributed training of large models. Unlike other systems such as Megatron (Shoeybi et al., 2020) and DeepSpeed (Smith et al.,

[1]NVIDIA, USA. Correspondence to: Anxhelo Xhebraj <axhebraj@nvidia.com>, Sean Lee <selee@nvidia.com>, Hanfeng Chen <hanfengc@nvidia.com>, Vinod Grover <vgrover@nvidia.com>.

Proceedings of the 8^{th} MLSys Conference, Santa Clara, CA, USA, 2025.

2022), model implementations do not have to commit to a concrete parallelization strategy. Instead, by building on top of GSPMD, parallelism is decoupled from the implementation and is introduced through lightweight sharding annotations. Additionally, JaxPP advances beyond SPMD by allowing arbitrary MPMD distributed dataflow in the pipeline parallelism dimension. We make the following contributions:

- We introduce a novel programming model that enables users to express pipeline parallelism seamlessly. The programming model does not require any user intervention to handle (potentially non-adjacent) communication across pipeline stages.
- We present a task-graph implementation that enables JaxPP to schedule tasks over a distributed mesh of devices, infer communication among them, and perform resource management tasks such as allocation and buffer deletion.
- We present JaxPP's single-controller MPMD runtime, that supports the execution of arbitrary user specified pipeline schedules.
- We demonstrate the benefits of our design by highlighting performance characteristics of JaxPP and compare it against state of the art alternatives on practical large-scale training benchmarks.

The rest of the paper is structured as follows. In Section 2, we motivate our system by describing the limitations of existing parallelization "interfaces" when applied to pipeline parallelism. Then, we give an overview of JaxPP (Section 3) and describe its runtime (Section 4). We extensively evaluate JaxPP's performance characteristics and compare against other state of the art systems (Section 5). We conclude by highlighting related work (Section 6) and final remarks (Section 7).

2 Motivation: User-Driven Partitioning Beyond SPMD

Scaling deep learning model training to large distributed clusters of devices requires a combination of several parallelization techniques. For example, the training of Llama 3 models (Dubey et al., 2024) used the combination of data parallelism, tensor parallelism, pipeline parallelism, and context parallelism. Implementing these parallelization strategies manually requires substantial effort.

To simplify this process, a few libraries and frameworks enable the post-hoc parallelization of existing models without major rewrites. Notably, recent work such as GSPMD (Xu et al., 2021), which is integrated within JAX (Bradbury et al., 2018), and PartIR (Alabed et al., 2024) introduce programming models that decouple model implementation from parallelization strategies, making distributed training more accessible. We now briefly describe JAX's programming model for parallel computations and explore its limitations for pipeline parallelism.

2.1 SPMD Parallelization Through Named Axes

To parallelize an array computation in JAX, we arrange a set of devices in a logical mesh, which is a multi-dimensional array of non-repeating devices. The mesh shape and device order can be arbitrary, but it is usually such that dimensions of the mesh correspond to a particular communication bandwidth. For example, 4 nodes with 8 GPUs each could have a mesh shape of (4, 8) where each row corresponds to the 8 devices present on each node. Therefore communication between devices within the same row is faster over communication between devices across different rows. The mesh dimensions can also be named, for example, `[("data", 4) ("model", 8)]`.

Given a mesh, an array can be *sharded* (or partitioned) by mapping some axes of the array to some axes of the mesh. If an array axis is not mapped to any mesh axis, then that axis is replicated across the remaining dimensions of the mesh. The snippet below shows shardings of the two-dimensional array `A` arising due to different partitioning specifications.

```
# mesh.shape=[("data", 4) ("model", 8)]
# A.shape=(n, m)
shard(A, (None   , "model")) # col (n  , m/8)
shard(A, ("data" , None))    # row (n/4, m  )
shard(A, ("data" , "model")) # 2D  (n/4, m/8)
```

The examples above mentioning only one axis of the mesh, will lead to the replication of the sharded tensor on the unmentioned axis of the mesh. For example, in the first case that mentions only the `"model"` axis, `A` is replicated across the 4 `"data"` groups and sharded column-wise within each of them.

Instead of specifying concrete mesh axes in model definitions, usually *logical axis names* (*named axes*) are used. This allows exploring several parallelization strategies with different mesh shapes without any changes to the model implementation. Axis names must be unique for the axis of one array, but can be shared across multiple arrays.

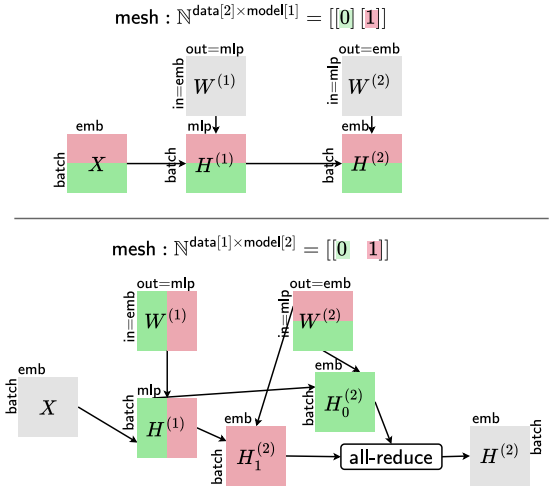
Figure 1a shows the definition of a Feed-Forward Network (FFN) using logical axis names. Note that the function has no collectives in its implementation and can be run on a single device. The function takes as argument a 2D input X with `batch` and `emb` (embedding) dimension and maps it to the 2D output $H^{(2)}$ with same logical axis names. The body of the function consists of the application of two parameters $W^{(i)}$ interleaved with an activation. Note that, while the input and output share logical axis names, their sizes can differ. Figure 1b specifies on what mesh dimension each

```
@shard(("batch", "emb"), ("emb", "mlp"), ("mlp", "emb"))
def ffn(X, W(1), W(2)):
  H(1) = relu(XW(1))
  H(1) = shard(H(1), ("batch", "mlp"))
  H(2) = H(1)W(2)
  return shard(H(2), ("batch", "emb"))
```

(a) Model implementation with named axes

partitioning = [
  batch ▷ data,
  mlp ▷ model
]

(b) Partitioning specification



(c) Different parallelism instantiations depending on the mesh shape

Figure 1. **Configurable Parallelism Through Named Axes in JAX** (Bradbury et al., 2018) Top left (1a): Model implementation where array axes are annotated with logical names. Bottom left (1b): Partitioning specification mapping logical axis names to mesh axes. Right (1c): Two parallel instantiation, data-parallel on the top with mesh shape `[("data", 2) ("model", 1)]` while a tensor-parallel implementation at the bottom when the mesh shape is `[("data", 1) ("model", 2)]`.

logical axis name should be sharded on. The parallelization of the computation is still undefined and will depend only on the concrete instantiation of the device mesh.

Data Parallelism (DP) replicates the model weights across devices while partitioning the batch among them. Figure 1b (Top) shows the corresponding mesh instantiation of shape `[("data", 2) ("model", 1)]` to achieve this. Since all axes of the weights $W^{(i)}$ are either unbound (`emb`) or are mapped to a mesh dimension of size 1 (`mlp` mapped to `model`), the weights are replicated (shown as gray blocks) across the two devices. For training, each "replica" computes the gradients with respect to the local batch. The gradient computation, which is not shown in the figure, requires contracting the activations on the batch dimension, leading to an `all_reduce` operation (replicas synchronize their gradients). This parallelization strategy allows training over larger "global" batch sizes, potentially leading to faster convergence.

Tensor Parallelism (TP) (Shoeybi et al., 2020) partitions weights of an individual layer over multiple devices. This allows running large models for which the training state does not fit into a single device. However, depending on the operations performed on the weights, collective operations may be required to complete the computation. XLA inserts them automatically as needed. The "Megatron-style" parallelization strategy corresponding to the mesh shape `[("data", 1) ("model", 2)]` is shown in Figure 1c (Bottom). The subscript of a variable (e.g., 0 in H_0) denotes the *shard*. Because the `mlp` axis is bound to the `model` axis of the mesh, which is composed of 2 devices, $W^{(1)}$ is partitioned on the output dimension (column-wise) while $W^{(2)}$ is partitioned in the input dimension (row-wise). The input and output of the function are replicated. The collective necessary for performing the parallel computation is inserted implicitly by XLA's SPMD partitioner. The second matrix-multiply operation $H^{(1)}W^{(2)}$ requires only one final all-reduce to compute the replicated output.

It is possible to combine DP and TP by defining a larger mesh such as `[("data", 4) ("model", 8)]`. In this scenario, 32 GPUs are split into 4 DP "groups" each constituted by 8 TP groups. Weights are replicated across the 4 DP groups and sharded across the 8 TP groups within each DP group. Similarly the batch is sharded across the 4 DP groups and then each shard is replicated within each TP group.

Finally, this programming model also allows more complex parallelism strategies such as Expert Parallelism (EP) (Lepikhin et al., 2020), where expert weights and intermediate activations are sharded and multiplied in parallel.

2.2 Limitation of SPMD: Pipeline Parallelism

All the parallelism strategies described so far fall into the SPMD category. Under this model, a single program is compiled and executed across multiple devices, each processing distinct input shards. This approach enables scalable deployment across thousands of partitions and simplifies

scheduling, especially for collective operations. However, for larger scale of number of devices, collectives necessary in the SPMD model can greatly harm performance.

Pipeline Parallelism (PP) offers an alternative by introducing temporal parallelism, dividing the computation graph into *stages*, and performing *gradient accumulation* over smaller partitions of the batch, called *microbatches*. From here, we use the term *actor* to refer to a group of devices.

2.2.1 The Importance of Pipeline Schedules

The first successful application of pipeline parallelism with synchronous gradient application was demonstrated in GPipe (Huang et al., 2019). A neural network's layers are split into *stages*. For each stage there is a forward computation and backward computation which must be scheduled in the same actor as the forward one. Each actor is assigned one stage, and iteratively executes the forward computation for each microbatch by, first saving potential activations needed for the backward computation, and then sending the output to the next actor. At the end of all microbatches, an actor receives the gradients of the activations from the succeeding actor and executes each backward computation. Finally, the accumulated gradients are used to update the model weights and optimizer parameters at the end of the training step. Since the activations of each microbatch have to be stored until the corresponding backward computation, memory usage in GPipe is proportional to the number of microbatches. Therefore, GPipe is usually combined with activation rematerialization (Chen et al., 2016).

Later works such as 1F1B (Narayanan et al., 2019) and Interleaved 1F1B (Narayanan et al., 2021) improved both memory usage and throughput of PP. Figure 2 shows the difference between the two schedules. The key realization is that the various stages of different gradient accumulation iterations can be scheduled arbitrarily as long as data dependencies are honored. Therefore, a gradient accumulation loop can be implemented in various ways with different *schedules* which describe the order and on which actor each stage's computation (or *task*) is run.

The 1F1B schedule shortens the lifetime of activations by eagerly scheduling the execution of backward stages. As a result, memory requirements become proportional to the number of stages instead of the number of microbatches, potentially translating to a $2\times$–$3\times$ reduction in activation memory. This increased memory availability also improves throughput since more activations can be stored and not rematerialized. Consequently, the end-to-end time of a training step can be reduced by 20% as we explain in Section 5.3.

Interleaved 1F1B further reduces idling time by assigning multiple stages to each actor. The number of stages per actor is referred to as the degree of *circular repeat*. As the degree of circular repeat increases, stages become smaller, enabling finer-grained scheduling. This approach improves throughput, but introduces additional communication overhead.

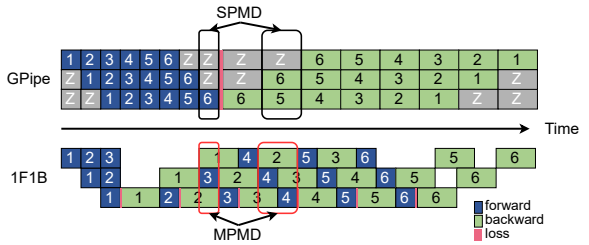


Figure 2. Comparison between GPipe and 1F1B. In GPipe, at any time, all pipeline-parallel groups perform the same computation. Bubbles are implemented as redundant discarded computation (gray Z blocks). In 1F1B, all groups perform different computations.

2.2.2 SPMD Encoding of Pipeline Parallelism

Xu et al. presented a clever encoding of pipeline parallelism as sharding in GSPMD. Assuming that all the stages have the same dataflow graph and input and output shapes (i.e. stages are *homogeneous*), it is possible to "*stack*" the weights of the layer and perform all stages in parallel by sharding the weights on the new leading dimension. Then, the same computation is applied to a sharded "state" buffer for a number of times in a loop, until all the microbatches have been processed. During the pipeline bubble iterations, idling actors participate in the computation (gray Z blocks in Figure 2) discarding the iteration's result. After the pipeline loop on the stacked layers, the outputs are used to compute the loss of the full batch.

Besides being unsuitable for models with non-homogeneous stages, GSPMD encoding can also negatively impact performance in the following ways: homogeneous stages forbid using different rematerialization strategies across stages and strict synchronization at each loop iteration forces all processes to wait for stragglers.

JAX's automatic differentiation (autodiff) generates a corresponding loop for the backward pass consuming the activations in reverse order. After SPMD partitioning, the generated code corresponds to the GPipe schedule. There is no way for the user to *control* the scheduling of the sections of the gradient accumulation loop, forgoing potential performance benefits described in Section 2.2.1. Although we do not preclude the existence of some program transformations that could encode 1F1B in the SPMD paradigm under further assumptions, such transformations would be inadequate. They would fail to respect the true essence and flexibility of pipeline schedules, which clearly necessitate a MPMD paradigm, where at any time different actors perform different stages of the loop (Figure 2 bottom).

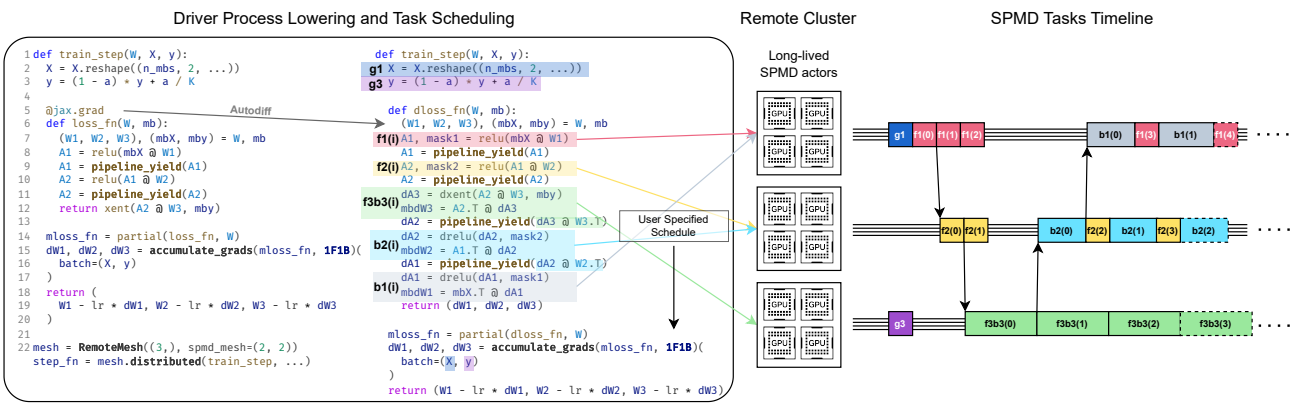


Figure 3. System Overview. The left box shows the code in the driver process describing the computation and annotating pipeline stage boundaries. Auto-differentiation produces additional stages corresponding to the "backward" computations for the gradients. The user specifies a mapping of stages to SPMD actors and a schedule for the loop. Each call to the `step_fn` function schedules tasks

3 JaxPP Overview

We now describe JaxPP, a compiler and runtime for running distributed MPMD computations. JaxPP extends JAX's user-driven SPMD parallelism by introducing task-based temporal parallelism. To achieve this, it addresses two key challenges: (1) User-scheduled gradient accumulation: we introduce a familiar loop construct that integrates user-defined schedules seamlessly in existing code; (2) Asynchronous task execution: we develop a runtime capable of efficiently executing distributed task graphs in parallel.

Figure 3 gives an overview of the compilation and runtime components of the system. The user's specification of `train_step` differs only slightly from a standard training step in JAX without pipeline parallelism. Additionally, the code is updated to perform gradient accumulation over the microbatches with `accumulate_grads` (L14) and the model is annotated with auto-differentiable `pipeline_yield` calls marking the end of the current stage (L9, L11). The `distributed` function traces the auto-differentiated training step into an intermediate representation called Jaxpr,[1] which is transformed and split into multiple tasks by the driver process. These tasks are then lowered and sent to the respective SPMD actors as specified by the schedule to be compiled and run on the *remote* devices. The remote devices are allocated by the driver process by instantiating a `RemoteMesh`. In the example displayed, 3 SPMD actors are provisioned, each with 4 devices configured in a SPMD mesh shape of `(2, 2)`. JaxPP attempts to group devices so that those assigned to an SPMD actor are connected through a high-bandwidth interconnect. Each task is lowered and compiled by XLA, leading to the same exact SPMD parallelization strategy the user would expect as described in Section 2.1 within each task. JaxPP infers tasks for all communication and resource management needed to perform the program's execution, such as send and receive operations and deallocation of intermediate buffers. All these tasks are fused into a single MPMD "program", so that at each call of the returned `step_fn` function, all the tasks can be dispatched into a single RPC call per SPMD actor.

In this section we describe key features of JaxPP that enable MPMD pipeline parallelism.

3.1 Gradient Accumulation Loop

Figure 4 highlights in teal the changes required to adopt JaxPP in an existing JAX training program. The model definition can leverage JAX's sharding annotations as shown in Section 2.1. The code is updated to implement the gradient accumulation loop over the microbatches with `accumulate_grads` (Section 3.1). The argument to `accumulate_grads` is a function (`microbatch_grads`) that given one microbatch produces the gradients and additional metrics of that microbatch. Semantically `accumulate_grads` will call `microbatch_grads` on each microbatch in `batch` and sum the gradients and collect the loss from each iteration, equivalently to the code below.

```
grads = zeros_like(state.params)
loss = []
for i in range(batch.shape):
  mugrads, muloss = microbatch_grads(batch[i])
  grads += mugrads
  loss.append(muloss)
```

The API is configured by default to implement the addition and concatenation operator on each iteration's output with the loop state. Internally, the API lowers to a proper structured "for loop" with an explicit state that is updated in the loop body. This API restriction is intentional to ensure that the provided loop body does not create dependencies between earlier stages of the current iteration of the loop

[1] https://jax.readthedocs.io/en/latest/jaxpr.html

with later stages of the previous iteration.

During compilation the gradient accumulation loop is "unrolled" into a task graph that is then scheduled and run on the remote devices.

```
@shard( () , ("emb", "mlp"), ("mlp", "emb"), ())
def ffn(X, W^(1), W^(2)):
  H^(1) = self.act(XW^(1))
  H^(1) = shard(H^(1), ("batch", "mlp"))
  A^(1) = jaxpp.pipeline_yield(H^(1))
  H^(2) = A^(1)W^(2)
  return shard(H^(2), ("batch", "emb"))

def loss_fn(...):
  # calls ffn
  ...

model, lr_scheduler, state = ...

def train_step(state, batch):
  def microbatch_grads(mubatch):
    # mubatch.shape=rest
    muloss, mugrads = jax.value_and_grad(loss_fn)(
      state.params, mubatch
    )
    # muloss.shape=()
    return mugrads, muloss
    #      +       ||

  schedule = _1F1B(stages=2)

  # batch.shape=(n_mbs, *rest)
  grads, loss = (
    jaxpp.accumulate_grads(microbatch_grads, schedule)
      (batch)
  )
  # loss.shape = (n_mbs,)
  new_state = state.apply_gradients(grads=grads)

  return new_state, loss

mesh = RemoteMesh((2,), spmd_mesh=(2, 2))
jit_train_step = mesh.distributed(
  train_step,
  in_shardings=(state_sharding, batch_sharding),
  out_shardings=(state_sharding, None)
)

for batch in dataset:
  state, loss = jit_train_step(state, batch)
```

Figure 4. Training loop in JaxPP

3.2 Stage Marking

An operation like `a, b = pipeline_yield(a, b, name='s0')` defines a corresponding "logical stage" (e.g., named computation), comprising of all the computations that the arguments (`a` and `b`) *depend on* and that have not been scheduled in any other stage. Figure 5 shows a snippet defining

```
z = ...
x = a + b * z
y = c + d * z
x = pipeline_yield(x, 's0')
y = pipeline_yield(y, 's1')
w = y + 1
...
return x, w, z ...
```

```
@task
def s0(a, b):
   z = ...
   x = a + b * z
   return (z, x)

@task
def s1(c, d, z):
   y = c + d * z
   w = y + 1
   return w

z, x = s0(a, b)
w = s1(c, d, z)
...
```

Figure 5. Program using `pipeline_yield` (left) and stages inferred by JaxPP (right)

two stages using `pipeline_yield` and the resulting stages inferred by JaxPP. Semantically, `pipeline_yield` returns the argument(s) as result(s) and is only a marker for JaxPP to infer corresponding stages.

The key idea is that stages are defined purely by their data dependencies: `y`'s computation is scheduled in `s1` although its definition appears earlier than the first `pipeline_yield` in the initial program; `z` is defined on the stage that uses it first (`s0`), then it's returned and passed as an argument to `s1`. We highlight that `pipeline_yield` does not necessarily mark the outputs of a stage, but only an *approximate* definition of a stage's end. The computation of `w` is placed in stage `s1` as it depends on `y`, which is placed in `s1`, and doesn't define any computation passed to another `pipeline_yield`. To schedule two independent computations such as the ones for `x` and `y`, in the same stage, it's sufficient to perform `x, y = pipeline_yield(x, y)`.

Users can forego the use of `pipeline_yield` and define stages equivalently as separate functions, writing directly the code shown in Figure 5 (right). JaxPP might include other computations on such stages as deemed ideal for performance. For example, in the presence of residual connections, gradient merging operations produced by auto-differentiation will be fused into the task that produces the latest operand.

We've found that the dependency-based definition of stages through `pipeline_yield` is less disruptive when incorporating JaxPP into existing codebases as it does not require refactoring model code into separate functions.

Both `task` and `pipeline_yield` are trivially auto-differentiable and compose with many other JAX transformations such as `vmap`. This enables using different rematerialization strategies for specific subcomputations or stages exactly as done in JAX[2] and shown below.

[2] https://docs.jax.dev/en/latest/jep/11830-new-remat-checkpoint.html#user-customizable-rematerialization-policies

```
def layer(...): ...
...

for W, policy in [(W1, p1) , (W2, p2)]:
  # JAX rematerialization API
  h = jax.checkpoint(layer, policy)(W, h)
  h = pipeline_yield(h)
```

3.3 Placement Inference

Given the `pipeline_yield` annotations and the accumulation loop, JaxPP automatically infers data placement for inputs and outputs of the `train_step` function.

In order for pipeline parallelism to work efficiently we have to ensure that each computation is scheduled at the right pipeline execution unit where the data are "pinned" while at the same time minimizing communication across actors. We assume that the loop schedule maps backward computations to the same actor of the corresponding forward computation. For example if weights W_1 and W_2 are placed on a specific actor then all the computation corresponding to the backward computation for the gradients of W_1 and W_2 must be scheduled on the same actor.

We use the following propagation heuristic to schedule operations on a task. First, a task is formed for each `pipeline_yield` operation, comprising of all computations it depends on. Then the remaining computations that are not dependencies of any `pipeline_yield` operation are placed on the same task of their operands or a new task. In the loop body we do not allow any computation replication, and instead each operation can be assigned to only one task. This step also infers the placement of inputs and outputs of the `accumulate_grads` loop.

Then input placement is propagated to the computation preceding the pipeline loop, potentially replicating computation and ensuring that the inferred placement does not overwrite the current placement and similarly loop output placement are propagated to the computation after the loop.

3.4 Weight Sharing and Gradient Accumulation

In the presence of weight sharing where the same weight is used across multiple stages as with tied embeddings in Transformer models, multiple partial gradients are computed from each use which are then added to form the full gradient, e.g., $g = ((g_1 + g_2) + g_3) + \dots$. A naive scheduling of such operations would lead to sends and receives of multiple partial gradients which, for embedding tables, can easily consist of several Gigabytes of data.

JaxPP implements a loop commuting pass which substitutes the carried state of the cumulative total gradient g with a carried state of the partial gradients $g_1, g_2, g_3, \dots$ and a final addition operation. This corresponds to the following rewrite rule.

$$g = \sum_{i=1}^{\#\text{microbatches}} \left(g_1^{(i)} + g_2^{(i)} + \dots \right)$$
$$\rightsquigarrow \left(\sum_{i=1}^{\#\text{microbatches}} g_1^{(i)} \right) + \left(\sum_{i=1}^{\#\text{microbatches}} g_2^{(i)} \right) + \dots$$

4 Scheduling and Runtime

In the previous section we introduced key Jaxpr transformations and user functions in JaxPP. We now describe the runtime architecture and how tasks are dispatched.

4.1 Architecture

The user program containing the code for training is run in a single Python process, we call driver or controller, on a host possibly co-located in the datacenter where training nodes reside. The controller is responsible for the tracing and transformations described in the previous section. Additionally the controller allocates stateful actors managing one or multiple devices possibly spanning multiple hosts. We use Ray (Moritz et al., 2018) for Remote Procedure Calls (RPCs) and orchestrating worker processes running XLA computations. We implement a custom on-device object store on each actor for storing sharded device buffers. Communication is handled using NCCL P2P operations.

4.1.1 Single-Controller MPMD

Using a single-controller model is not strictly necessary for an MPMD implementation of pipeline parallelism, but it offers several key benefits, such as easier scaling and reduced code complexity. In contrast, a multi-controller approach can be implemented by replicating the controller logic across all processes, allowing each process to generate its own local tasks. However, this method requires users to carefully manage code sections that need to run in a single-threaded context and manually dispatch processes to ensure correct placement—for example, keeping model-parallel groups on the same host while distributing pipeline-parallel groups across different hosts. With a single-controller model, users can scale from a single-device setup to multiple devices across hosts with minimal code changes. The primary effort involves annotating the training step function, simplifying the transition without extensive rewrites.

4.2 Task Scheduling

A user can specify a loop schedule declaratively by providing a list of tasks for each actor describing the iteration of the loop run, its type (forward or backward) and the stage index. This is shown in the listing below.

```
[
  [ # actor_1
    Task(i=0, ty='fwd', stage=0),
    Task(i=1, ty='fwd', stage=0),
    Task(i=0, ty='fwd', stage=2), ...
  ],
  [ # actor_2
    Task(i=0, ty='fwd', stage=1),
    Task(i=1, ty='fwd', stage=1),
    Task(i=0, ty='fwd', stage=3), ...
  ],
]
```

JaxPP builds a task graph based on task placement and task dependencies to then infer allocation, send, and receive operations. Care has to be taken when generating the local task schedule for each actor, especially in the generation of send and receive operations. This is because communication primitives, although asynchronous, still require send and receive operations to have matching receive and send operations in the same order respectively among the communicating processes to prevent potential deadlocks. Therefore, simply iterating over each local task of an actor and performing receive operations for the non-local task operands, executing the task and sending the results immediately as shown in Figure 6 can potentially result in deadlock.

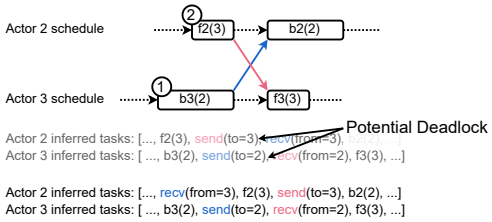


Figure 6. Inference of send and receive operations based on uses and definitions in the task graph.

Instead, JaxPP iterates over the tasks in their topological order and schedules asynchronous send and receive pairs immediately after the corresponding task has produced the data to be communicated. In the example above, after scheduling `b3` on actor numbered 3, JaxPP immediately schedules a send and the corresponding receive on the receiving actor. Since receive operations are asynchronous, the computation of task `f2(3)` is overlapped with the potential prefetching of the data from actor 3 which is used only later in task `b2(2)`.

4.3 Buffer Deletion

After generating the local task schedule for each actor, a buffer liveness pass inserts deletion operations for intermediate buffer. A buffer that is sent to an actor is tentatively deleted if the corresponding send operation has completed, otherwise it is tracked into a "pending deletions" queue for later reclamation. Each scheduled deletion operation checks this queue and deletes previously pending deletions.

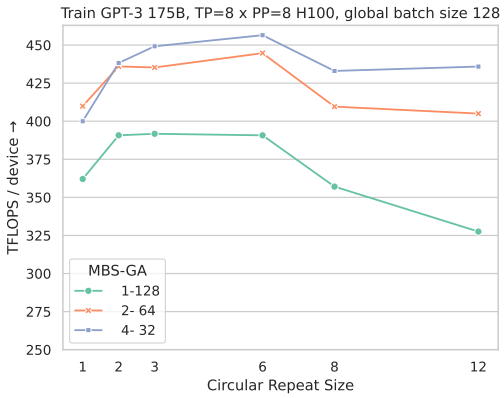


Figure 7. Performance of GPT-3 175B training on 64 GPUs with global batch size of 128 instances across various configurations for interleaving/circular repeat and microbatch size.

4.4 Task fusion

A direct implementation of task dispatch on the driver would perform a separate remote procedure call. Multiple round trips of "control" would lead to poor utilization, especially when running in a loop. The `distributed` annotation fuses all task dispatches into a single RPC call per actor. All the coordination between multiple actor is resolved by send and receive dependencies only.

5 Evaluation

In this section, we analyze important performance characteristics of pipeline parallelism as implemented in JaxPP (Section 5.1) and evaluate the performance gains achieved by JaxPP in comparison to other systems that support large language model training with various parallelism strategies (Section 5.2). We conducted our experiments on NVIDIA EOS cluster (eos, 2024) which is equipped with NVIDIA DGX H100 with the InfiniBand NDR400 interconnect. Each node consists of 8 H100 GPUs, each with 80 GB of memory. We evaluate JaxPP on the training of GPT-3 175B (Brown et al., 2020) and Llama2 70B (Touvron et al., 2023) at BF16 precision.

5.1 Performance Characteristics

In this section, we discuss the performance characteristics of JaxPP for specific configurations, relating them to the design of the system and its potential overheads.

5.1.1 Interleaving and Dispatch Overhead

An important distinction of JaxPP over other plain JAX implementations is that JaxPP splits the training step computation into multiple XLA SPMD tasks, e.g., forward and backward computations for each stage. This is necessary to

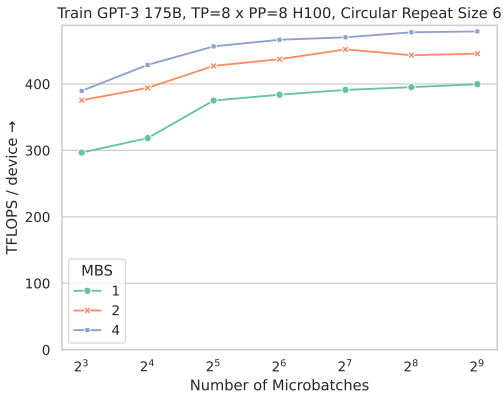


Figure 8. Performance of GPT-3 175B training on 64 GPUs with circular repeat size of 6 and various combinations of gradient accumulations and microbatch sizes.

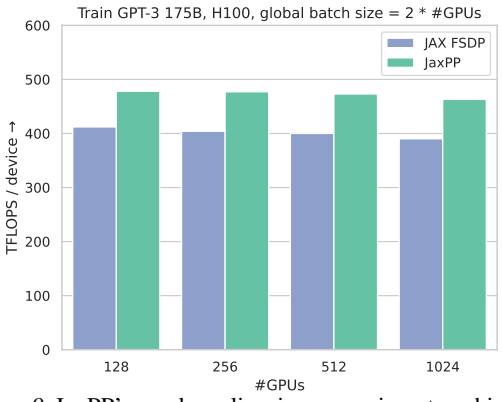


Figure 9. JaxPP's weak scaling in comparison to a highly optimized JAX FSDP implementation.

implement the various pipeline schedules. However, it can incur dispatch overheads. Such overheads can especially be exacerbated when using configurations that try to reduce pipeline bubbles, such as: (1) slicing the dataflow graph into smaller stages and using interleaved schedules (Narayanan et al., 2021) (2) slicing the batch into smaller microbatch sizes resulting in more microbatches.

Smaller stages as in (1) increase the number of XLA asynchronous dispatches which have non-negligible cost if the device work dispatched is too small. Smaller microbatches as in (2) can lead to poorer kernel-level device utilization and increase the number of collectives, e.g., the kernel time t_2 for one microbatch of size 2 can be smaller than kernel time for 2 microbatches of size 1 each taking t_1 ($t_2 < 2t_1$).

Figure 7 explores this tradeoff. As shown in the picture increasing the number of circular repeats of stages, leading to smaller tasks, improves for all cases up to the point when the tasks become too small and XLA dispatch overheads emerge and P2P latencies start becoming non-negligible. Increasing the microbatch size increases the bubble time but at the same time reduces the number of collectives per loop iteration, overall improving performance.

5.1.2 Utilization Tradeoff

Given a model size and a pipeline parallel configuration, it is possible to increase the overall global batch size either by accumulating over more microbatches or scaling in purely data-parallel fashion. Increasing the number of microbatches is beneficial to minimize the pipeline bubble. However, given a fixed target number of tokens to train on, it increases end-to-end training time since more work is done iteratively instead of being parallelized. At the same time, scaling DP at low utilization is not cost effective. Figure 8 shows the utilization achieved by JaxPP at different numbers of microbatches for multiple microbatch sizes.

5.1.3 Scalability

In order to test the scalability of JaxPP, we conducted weak scaling experiments on the GPT-3 175B model by increasing the global batch sizes linearly from 128 to 2048, with 32 microbatches, doubling the number of GPUs. 4-way tensor parallelism and 2-way data parallelism was enabled within each node containing 8 GPUs, and 8-way pipeline parallelism was enabled across each group of 8 nodes, using Interleaved 1F1B as the schedule and a circular repeat of size 6. We instantiate a JaxPP actor per node.

As illustrated in Figure 9, JaxPP effectively scales GPT-3 175B training from 64 to 1024 GPUs, achieving a 98.33% weak scaling efficiency. This performance improves over the 93.97% efficiency demonstrated by a highly optimized system utilizing Fully-Sharded Data Parallelism (FSDP) with JAX. JaxPP exceeds the scaling efficiency of JAX FSDP while also delivering higher throughput and lower end-to-end latency.

5.2 Training Performance

In this section, we compare the performance of model training in JaxPP against the SPMD-based pipeline parallelism solution in JAX and the state-of-the-art implementation found in NeMo (Harper et al.). This comparison aims to validate our claim that JaxPP overcomes the limitations of SPMD-based pipeline parallelism without compromising performance. Additionally, we discuss how JaxPP achieves performance gains in certain scenarios and explore potential avenues for further improving its performance.

As depicted in Figure 10, when training GPT-3 175B on 16 DGX H100 nodes (128 GPUs), JaxPP is 51.2% faster than SPMD pipeline parallelism, achieving 478 TFLOPS/device, while being more expressive and requiring 1K fewer lines of user code. Moreover, JaxPP improves throughput by 1.16× over JAX's FSDP. JaxPP achieves 95.6% throughput of NeMo's pipeline parallelism while being entirely

model-agnostic. The comparison is similar when training Llama2 70B on 8 DGX H100 nodes (64 GPUs), demonstrating JaxPP effectiveness at smaller scales. JaxPP uses upstream JAX and XLA with no custom kernels except for the attention APIs from cuDNN (Chetlur et al., 2014).

5.2.1 *Pipeline Parallelism and FSDP*

Now that we've shown that PP improves performance over FSDP and JaxPP considerably simplifies the adoption of PP, we describe the relation between FSDP and PP.

One limiting factor of pure FSDP is that when scaling on a large number of GPUs, the batch size for training has to scale proportionally (e.g., $8,192 \cdot k, k \in \mathbb{N}$ for scaling to $8,192$ GPUs). This has been described in detail by Lamy-Poirier (2022). Therefore FSDP is usually combined with other parallelization strategies such as TP and others, which partition activation dimensions other than the batch one. However, depending on the parallelization strategy, overlapping multiple collectives can prove difficult.

Pure FSDP is not usually combined with PP. However, FSDP-like behavior can be used to enhance PP. For example, the optimizer state and weights can be sharded across all processes. Before the pipeline loop, weights are cast to the corresponding datatype and an all-gather is performed to materialize the weights on each device. During the loop iterations, weights and gradients are partially replicated over the data-parallel dimension. At the end of the gradient accumulation loop, gradients are reduce-scattered to obtain the full gradients and the optimizer state and parameter shards are updated.

Finally, PP is beneficial also in cases where the amount of GPUs is limited, but we'd like to train on a large batch size (e.g., 64 GPUs and GBS $= 1024$). By accumulating the gradients over multiple microbatches, it's possible to train over a larger effective batch size while keeping the memory footprint constant.

5.2.2 *Comparison to NeMo*

Most parts of the parallelization configuration between NeMo and JaxPP are similar, differing in the following ways: (1) NeMo uses tensor sequence parallelism (Korthikanti et al., 2023) for normalization layers while JaxPP shards the activation dimension. (2) NeMo uses "Async TP" (Collective Matmul) (Wang et al., 2022). A more important distinction between the two is that NeMo is a collection of high-performance model *implementations* while JaxPP, by building on top of JAX and XLA, comprises of a set of compiler transformations and runtime for general Linear Algebra programs. Therefore, in JaxPP, many optimizations such as overlapping computation with communication, general scheduling, and fusion of operations are handled by XLA's compiler, making JaxPP a more general approach to pipelining arbitrary computations.

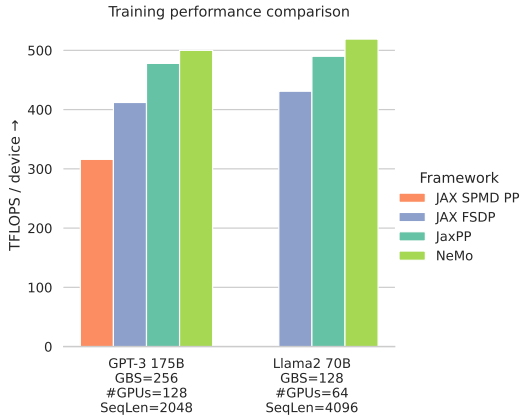


Figure 10. Performance comparison between SPMD pipeline parallelism, JaxPP, and NeMo on GPT-3 175B and Llama2 70B.

5.3 Performance Breakdown

To understand the sources of performance gains achieved by JaxPP over SPMD pipeline parallelism on GPT-3 175B, we present Figure 11. The most significant factor is the rematerialization cost. SPMD pipeline parallelism employs the GPipe schedule, which has high memory demands, whereas JaxPP utilizes the Interleaved 1F1B schedule, which requires less memory. This difference impacts the need for rematerialization, subsequently affecting the overall training step time by $\approx 20\%$. Additionally, JaxPP further reduces overhead by overlapping point-to-point send and receive operations, in contrast to their synchronous counterpart.

6 Related Work

There are numerous works to facilitate scaling the training of large models (Shoeybi et al., 2020; Rasley et al., 2020;

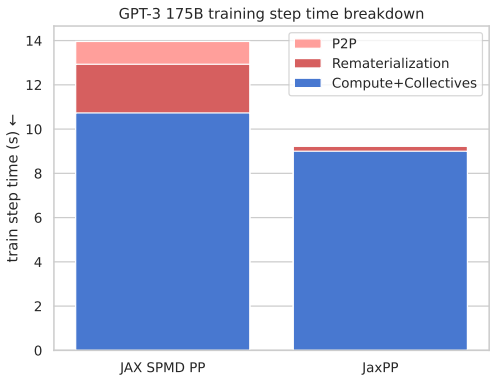


Figure 11. Overhead of JAX SPMD PP compared to JaxPP. Rematerialization cost and asynchronous point-to-point send and receive operations account for the majority of the performance differences.

System	GBS	GA	GPUs	PP	TP	DP	FSDP	Step Time (s)	TFLOPS / device
GPT-3 175B — BF16 — Sequence Length 2048									
JaxPP	256	32	128	8	8	2	1	9.22	478
	512	32	256	8	8	4	1	9.24	477
	1024	32	512	8	8	8	1	9.32	473
	2048	32	1024	8	8	16	1	9.37	470
JAX FSDP	256	1	128	1	1	1	128	10.70	412
	512	1	256	1	1	2	128	10.91	404
	1024	1	512	1	1	4	128	11.01	400
	2048	1	1024	1	1	8	128	11.30	390
JAX SPMD PP	256	128	128	16	4	2	1	13.96	316
NeMo	256	64	128	8	4	4	1	8.82	500
Llama2 70B — BF16 — Sequence Length 4096									
JaxPP	128	16	64	4	8	2	1	7.42	490
JAX FSDP	128	1	64	1	1	1	64	8.44	431
NeMo	128	32	64	4	4	4	1	7.02	519

Table 1. Training performance of JaxPP, JAX FSDP, JAX SPMD PP, and NeMo with GPT-3 175B and Llama2 70B. Different systems may use different combinations of various parallelism strategies based on their resource requirements and performance characteristics.

Liang et al., 2024; Jiang et al., 2024). Here we discuss systems that are closest to JaxPP and explain key design differences.

Alpa (Zheng et al., 2022) is a system for parallelizing large deep learning models, supporting pipeline parallelism. Similarly to JaxPP, Alpa implements an MPMD runtime on top of JAX/XLA and orchestrates the execution of SPMD tasks. Nonetheless, Alpa's main focus is automatically inferring the best optimal parallel strategy with respect to a cost model. JaxPP differs from Alpa in the following ways: it focuses on providing a flexible interface to let the users control parallelism instead of automating parallelism, no different from sharding annotations, greatly reducing compilation time; it does not fork JAX or XLA; it supports user-extensible stage execution mapping such as Interleaved 1F1B (Narayanan et al., 2021).

Pathways (Barham et al., 2022) is a single-controller distributed dataflow runtime for machine learning workloads. While some implementation details such as parallel dispatch and MPMD support are shared between Pathways and JaxPP, Pathways is fine-tuned to time sharing and multiplexing tasks, while JaxPP focuses on long-running training jobs, where resources such as memory, GPU, and interconnect bandwidth are fully allocated to the training job.

Finally, many recent works have proposed novel pipeline schedules for specific scenarios and new applications (Lamy-Poirier, 2022; Huang et al., 2024; Lin et al., 2024; Qi et al., 2024). Although we focused on practical applications of traditional schedules, JaxPP has all the features needed to support these novel schedules.

7 Conclusion and Future Work

We presented JaxPP, a system for implementing and efficiently running distributed dataflow computations. By extending the SPMD programming model of JAX with temporal parallelism, we showed that JaxPP provides a flexible environment for easily scaling training of deep learning models with pipeline parallelism. While the implementation presented here builds on top of JAX and XLA, the same core ideas can be leveraged to implement similar transformations as an MLIR (Lattner et al., 2021) dialect and build a MPMD runtime on other technologies.

Acknowledgements

We would like to acknowledge the helpful feedback and shepherding provided by our reviewers, which significantly improved our presentation. We extend our gratitude to Jonathan Dekhtiar, Nitin Nitin, Abhinav Goel, and Haixin Liu for their invaluable and continuous feedback. We're thankful to Oren Leung for his contributions in the initial design and prototype of JaxPP. Our thanks also go to Sangkug Lym, Tejash Shah, and Santosh Bhavani for their assistance in evaluating JaxPP. Finally, we are grateful to Shriram Janardhan, Jeremy Wilke, Deepak Narayanan, and Georg Stefan Schmid for their insightful and thought-provoking discussions.

References

EOS NVIDIA DGX SuperPod - NVIDIA DGX H100, 2024. URL https://www.top500.org/system/180239.

Alabed, S., Chrzaszcz, B., Franco, J., Grewe, D., Maclaurin, D., Molloy, J., Natan, T., Norman, T., Pan, X., Paszke, A., Rink, N. A., Schaarschmidt, M., Sitdikov, T., Swietlik, A.,

Vytiniotis, D., and Wee, J. PartIR: Composing SPMD Partitioning Strategies for Machine Learning, January 2024. URL http://arxiv.org/abs/2401.11202.

Barham, P., Chowdhery, A., Dean, J., Ghemawat, S., Hand, S., Hurt, D., Isard, M., Lim, H., Pang, R., Roy, S., Saeta, B., Schuh, P., Sepassi, R., Shafey, L. E., Thekkath, C. A., and Wu, Y. Pathways: Asynchronous Distributed Dataflow for ML. In *Proceedings of Machine Learning and Systems 2022, MLSys 2022, Santa Clara, CA, USA, August 29 - September 1, 2022*, 2022.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. JAX: Composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, volume 33, pp. 1877–1901. Curran Associates, Inc., 2020.

Chen, T., Xu, B., Zhang, C., and Guestrin, C. Training deep nets with sublinear memory cost. *CoRR*, abs/1604.06174, 2016. URL http://arxiv.org/abs/1604.06174.

Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. cudnn: Efficient primitives for deep learning, 2014. URL https://arxiv.org/abs/1410.0759.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., Schuh, P., Shi, K., Tsvyashchenko, S., Maynez, J., Rao, A., Barnes, P., Tay, Y., Shazeer, N., Prabhakaran, V., Reif, E., Du, N., Hutchinson, B., Pope, R., Bradbury, J., Austin, J., Isard, M., Gur-Ari, G., Yin, P., Duke, T., Levskaya, A., Ghemawat, S., Dev, S., Michalewski, H., Garcia, X., Misra, V., Robinson, K., Fedus, L., Zhou, D., Ippolito, D., Luan, D., Lim, H., Zoph, B., Spiridonov, A., Sepassi, R., Dohan, D., Agrawal, S., Omernick, M., Dai, A. M., Pillai, T. S., Pellat, M., Lewkowycz, A., Moreira, E., Child, R., Polozov, O., Lee, K., Zhou, Z., Wang, X., Saeta, B., Diaz, M., Firat, O., Catasta, M., Wei, J., Meier-Hellstern, K., Eck, D., Dean, J., Petrov, S., and Fiedel, N. PaLM: Scaling Language Modeling with Pathways, October 2022. URL http://arxiv.org/abs/2204.02311.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., and Houlsby, N. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL https://openreview.net/forum?id=YicbFdNTTy.

Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A., Goyal, A., Hartshorn, A., Yang, A., Mitra, A., Sravankumar, A., Korenev, A., Hinsvark, A., Rao, A., Zhang, A., Rodriguez, A., Gregerson, A., Spataru, A., Roziere, B., Biron, B., Tang, B., Chern, B., Caucheteux, C., Nayak, C., Bi, C., Marra, C., McConnell, C., Keller, C., Touret, C., Wu, C., Wong, C., Ferrer, C. C., Nikolaidis, C., Allonsius, D., Song, D., Pintz, D., Livshits, D., Esiobu, D., Choudhary, D., Mahajan, D., Garcia-Olano, D., Perino, D., Hupkes, D., Lakomkin, E., AlBadawy, E., Lobanova, E., Dinan, E., Smith, E. M., Radenovic, F., Zhang, F., Synnaeve, G., Lee, G., Anderson, G. L., Nail, G., Mialon, G., Pang, G., Cucurell, G., Nguyen, H., Korevaar, H., Xu, H., Touvron, H., Zarov, I., Ibarra, I. A., Kloumann, I., Misra, I., Evtimov, I., Copet, J., Lee, J., Geffert, J., Vranes, J., Park, J., Mahadeokar, J., Shah, J., van der Linde, J., Billock, J., Hong, J., Lee, J., Fu, J., Chi, J., Huang, J., Liu, J., Wang, J., Yu, J., Bitton, J., Spisak, J., Park, J., Rocca, J., Johnstun, J., Saxe, J., Jia, J., Alwala, K. V., Upasani, K., Plawiak, K., Li, K., Heafield, K., Stone, K., El-Arini, K., Iyer, K., Malik, K., Chiu, K., Bhalla, K., Rantala-Yeary, L., van der Maaten, L., Chen, L., Tan, L., Jenkins, L., Martin, L., Madaan, L., Malo, L., Blecher, L., Landzaat, L., de Oliveira, L., Muzzi, M., Pasupuleti, M., Singh, M., Paluri, M., Kardas, M., Oldham, M., Rita, M., Pavlova, M., Kambadur, M., Lewis, M., Si, M., Singh, M. K., Hassan, M., Goyal, N., Torabi, N., Bashlykov, N., Bogoychev, N., Chatterji, N., Duchenne, O., Çelebi, O., Alrassy, P., Zhang, P., Li, P., Vasic, P., Weng, P., Bhargava, P., Dubal, P., Krishnan, P., Koura, P. S., Xu, P., He, Q., Dong, Q., Srinivasan, R., Ganapathy, R., Calderer, R., Cabral, R. S., Stojnic, R., Raileanu, R., Girdhar, R., Patel, R., Sauvestre, R., Polidoro, R., Sumbaly, R., Taylor, R., Silva, R., Hou, R., Wang, R., Hosseini, S., Chennabasappa, S., Singh, S., Bell, S., Kim, S. S., Edunov, S., Nie, S., Narang, S., Raparthy, S., Shen, S., Wan, S., Bhosale, S., Zhang, S., Vandenhende, S., Batra, S., Whitman, S., Sootla, S., Collot, S., Gururangan, S., Borodinsky, S., Herman, T., Fowler, T., Sheasha, T., Georgiou, T., Scialom, T., Speckbacher, T., Mihaylov, T., Xiao, T., Karn, U., Goswami, V., Gupta, V., Ramanathan, V., Kerkez, V., Gonguet, V., Do, V., Vogeti, V., Petrovic, V., Chu, W., Xiong, W., Fu, W., Meers, W., Martinet, X., Wang, X.,

Tan, X. E., Xie, X., Jia, X., Wang, X., Goldschlag, Y., Gaur, Y., Babaei, Y., Wen, Y., Song, Y., Zhang, Y., Li, Y., Mao, Y., Coudert, Z. D., Yan, Z., Chen, Z., Papakipos, Z., Singh, A., Grattafiori, A., Jain, A., Kelsey, A., Shajnfeld, A., Gangidi, A., Victoria, A., Goldstand, A., Menon, A., Sharma, A., Boesenberg, A., Vaughan, A., Baevski, A., Feinstein, A., Kallet, A., Sangani, A., Yunus, A., Lupu, A., Alvarado, A., Caples, A., Gu, A., Ho, A., Poulton, A., Ryan, A., Ramchandani, A., Franco, A., Saraf, A., Chowdhury, A., Gabriel, A., Bharambe, A., Eisenman, A., Yazdan, A., James, B., Maurer, B., Leonhardi, B., Huang, B., Loyd, B., Paola, B. D., Paranjape, B., Liu, B., Wu, B., Ni, B., Hancock, B., Wasti, B., Spence, B., Stojkovic, B., Gamido, B., Montalvo, B., Parker, C., Burton, C., Mejia, C., Wang, C., Kim, C., Zhou, C., Hu, C., Chu, C.-H., Cai, C., Tindal, C., Feichtenhofer, C., Civin, D., Beaty, D., Kreymer, D., Li, D., Wyatt, D., Adkins, D., Xu, D., Testuggine, D., David, D., Parikh, D., Liskovich, D., Foss, D., Wang, D., Le, D., Holland, D., Dowling, E., Jamil, E., Montgomery, E., Presani, E., Hahn, E., Wood, E., Brinkman, E., Arcaute, E., Dunbar, E., Smothers, E., Sun, F., Kreuk, F., Tian, F., Ozgenel, F., Caggioni, F., Guzmán, F., Kanayet, F., Seide, F., Florez, G. M., Schwarz, G., Badeer, G., Swee, G., Halpern, G., Thattai, G., Herman, G., Sizov, G., Guangyi, Zhang, Lakshminarayanan, G., Shojanazeri, H., Zou, H., Wang, H., Zha, H., Habeeb, H., Rudolph, H., Suk, H., Aspegren, H., Goldman, H., Damlaj, I., Molybog, I., Tufanov, I., Veliche, I.-E., Gat, I., Weissman, J., Geboski, J., Kohli, J., Asher, J., Gaya, J.-B., Marcus, J., Tang, J., Chan, J., Zhen, J., Reizenstein, J., Teboul, J., Zhong, J., Jin, J., Yang, J., Cummings, J., Carvill, J., Shepard, J., McPhie, J., Torres, J., Ginsburg, J., Wang, J., Wu, K., U, K. H., Saxena, K., Prasad, K., Khandelwal, K., Zand, K., Matosich, K., Veeraraghavan, K., Michelena, K., Li, K., Huang, K., Chawla, K., Lakhotia, K., Huang, K., Chen, L., Garg, L., A, L., Silva, L., Bell, L., Zhang, L., Guo, L., Yu, L., Moshkovich, L., Wehrstedt, L., Khabsa, M., Avalani, M., Bhatt, M., Tsimpoukelli, M., Mankus, M., Hasson, M., Lennie, M., Reso, M., Groshev, M., Naumov, M., Lathi, M., Keneally, M., Seltzer, M. L., Valko, M., Restrepo, M., Patel, M., Vyatskov, M., Samvelyan, M., Clark, M., Macey, M., Wang, M., Hermoso, M. J., Metanat, M., Rastegari, M., Bansal, M., Santhanam, N., Parks, N., White, N., Bawa, N., Singhal, N., Egebo, N., Usunier, N., Laptev, N. P., Dong, N., Zhang, N., Cheng, N., Chernoguz, O., Hart, O., Salpekar, O., Kalinli, O., Kent, P., Parekh, P., Saab, P., Balaji, P., Rittner, P., Bontrager, P., Roux, P., Dollar, P., Zvyagina, P., Ratanchandani, P., Yuvraj, P., Liang, Q., Alao, R., Rodriguez, R., Ayub, R., Murthy, R., Nayani, R., Mitra, R., Li, R., Hogan, R., Battey, R., Wang, R., Maheswari, R., Howes, R., Rinott, R., Bondu, S. J., Datta, S., Chugh, S., Hunt, S., Dhillon, S., Sidorov, S., Pan, S., Verma, S., Yamamoto, S., Ramaswamy, S., Lindsay, S., Lindsay, S., Feng, S., Lin, S., Zha, S. C., Shankar, S., Zhang, S., Zhang, S., Wang, S., Agarwal, S., Sajuyigbe, S., Chintala, S., Max, S., Chen, S., Kehoe, S., Satterfield, S., Govindaprasad, S., Gupta, S., Cho, S., Virk, S., Subramanian, S., Choudhury, S., Goldman, S., Remez, T., Glaser, T., Best, T., Kohler, T., Robinson, T., Li, T., Zhang, T., Matthews, T., Chou, T., Shaked, T., Vontimitta, V., Ajayi, V., Montanez, V., Mohan, V., Kumar, V. S., Mangla, V., Albiero, V., Ionescu, V., Poenaru, V., Mihailescu, V. T., Ivanov, V., Li, W., Wang, W., Jiang, W., Bouaziz, W., Constable, W., Tang, X., Wang, X., Wu, X., Wang, X., Xia, X., Wu, X., Gao, X., Chen, Y., Hu, Y., Jia, Y., Qi, Y., Li, Y., Zhang, Y., Zhang, Y., Adi, Y., Nam, Y., Yu, Wang, Hao, Y., Qian, Y., He, Y., Rait, Z., DeVito, Z., Rosnbrick, Z., Wen, Z., Yang, Z., and Zhao, Z. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

Fedus, W., Zoph, B., and Shazeer, N. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity, June 2022. URL http://arxiv.org/abs/2101.03961.

Harper, E., Majumdar, S., Kuchaiev, O., Jason, L., Zhang, Y., Bakhturina, E., Noroozi, V., Subramanian, S., Nithin, K., Jocelyn, H., Jia, F., Balam, J., Yang, X., Livne, M., Dong, Y., Naren, S., and Ginsburg, B. NeMo: a toolkit for Conversational AI and Large Language Models. URL https://github.com/NVIDIA/NeMo.

Hooker, S. The Hardware Lottery, September 2020. URL http://arxiv.org/abs/2009.06489.

Huang, J., Zhang, Z., Zheng, S., Qin, F., and Wang, Y. DISTMM: Accelerating distributed multimodal model training. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pp. 1157–1171, Santa Clara, CA, April 2024. USENIX Association. ISBN 978-1-939133-39-7. URL https://www.usenix.org/conference/nsdi24/presentation/huang.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M. X., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., and Chen, Z. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In Wallach, H. M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E. B., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pp. 103–112, 2019.

Jiang, Z., Lin, H., Zhong, Y., Huang, Q., Chen, Y., Zhang, Z., Peng, Y., Li, X., Xie, C., Nong, S., Jia, Y., He, S., Chen, H., Bai, Z., Hou, Q., Yan, S., Zhou, D., Sheng, Y., Jiang, Z., Xu, H., Wei, H., Zhang, Z., Nie,

P., Zou, L., Zhao, S., Xiang, L., Liu, Z., Li, Z., Jia, X., Ye, J., Jin, X., and Liu, X. Megascale: Scaling large language model training to more than 10,000 gpus. In Vanbever, L. and Zhang, I. (eds.), *21st USENIX Symposium on Networked Systems Design and Implementation, NSDI 2024, Santa Clara, CA, April 15-17, 2024*, pp. 745–760. USENIX Association, 2024. URL https://www.usenix.org/conference/nsdi24/presentation/jiang-ziheng.

Korthikanti, V. A., Casper, J., Lym, S., McAfee, L., Andersch, M., Shoeybi, M., and Catanzaro, B. Reducing Activation Recomputation in Large Transformer Models. *Proceedings of Machine Learning and Systems*, 5, March 2023. URL https://proceedings.mlsys.org/paper_files/paper/2023/hash/e851ca7b43815718fbbac8afb2246bf8-Abstract-mlsys2023.html.

Lamy-Poirier, J. Breadth-First Pipeline Parallelism, November 2022. URL http://arxiv.org/abs/2211.05953.

Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., and Zinenko, O. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 2–14, 2021. doi: 10.1109/CGO51591.2021.9370308.

Lepikhin, D., Lee, H., Xu, Y., Chen, D., Firat, O., Huang, Y., Krikun, M., Shazeer, N., and Chen, Z. GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding, June 2020. URL http://arxiv.org/abs/2006.16668.

Liang, W., Liu, T., Wright, L., Constable, W., Gu, A., Huang, C.-C., Zhang, I., Feng, W., Huang, H., Wang, J., Purandare, S., Nadathur, G., and Idreos, S. Torchtitan: One-stop pytorch native solution for production ready llm pre-training, 2024. URL https://arxiv.org/abs/2410.06511.

Lin, Z., Miao, Y., Zhang, Q., Yang, F., Zhu, Y., Li, C., Maleki, S., Cao, X., Shang, N., Yang, Y., Xu, W., Yang, M., Zhang, L., and Zhou, L. nnScaler: Constraint-Guided parallelization plan generation for deep learning training. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 347–363, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/lin-zhiqi.

Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M. I., and Stoica, I. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 561–577, Carlsbad, CA, October 2018. USENIX Association. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/moritz.

Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. PipeDream: Generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 1–15, Huntsville Ontario Canada, October 2019. ACM. ISBN 978-1-4503-6873-5. doi: 10.1145/3341301.3359646. URL https://dl.acm.org/doi/10.1145/3341301.3359646.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., Phanishayee, A., and Zaharia, M. Efficient large-scale language model training on GPU clusters using megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, pp. 1–15, New York, NY, USA, November 2021. Association for Computing Machinery. ISBN 978-1-4503-8442-1. doi: 10.1145/3458817.3476209. URL https://dl.acm.org/doi/10.1145/3458817.3476209.

Pope, R., Douglas, S., Chowdhery, A., Devlin, J., Bradbury, J., Levskaya, A., Heek, J., Xiao, K., Agrawal, S., and Dean, J. Efficiently Scaling Transformer Inference, November 2022. URL http://arxiv.org/abs/2211.05102.

Qi, P., Wan, X., Huang, G., and Lin, M. Zero bubble (almost) pipeline parallelism. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=tuzTN0eIO5.

Rasley, J., Rajbhandari, S., Ruwase, O., and He, Y. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, pp. 3505–3506, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379984. doi: 10.1145/3394486.3406703. URL https://doi.org/10.1145/3394486.3406703.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism, March 2020. URL http://arxiv.org/abs/1909.08053.

Smith, S., Patwary, M., Norick, B., LeGresley, P., Rajbhandari, S., Casper, J., Liu, Z., Prabhumoye, S., Zerveas, G., Korthikanti, V., Zhang, E., Child, R., Aminabadi, R. Y., Bernauer, J., Song, X., Shoeybi, M., He, Y., Houston, M., Tiwary, S., and Catanzaro, B. Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model, February 2022. URL http://arxiv.org/abs/2201.11990.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Ferrer, C. C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., Fuller, B., Gao, C., Goswami, V., Goyal, N., Hartshorn, A., Hosseini, S., Hou, R., Inan, H., Kardas, M., Kerkez, V., Khabsa, M., Kloumann, I., Korenev, A., Koura, P. S., Lachaux, M.-A., Lavril, T., Lee, J., Liskovich, D., Lu, Y., Mao, Y., Martinet, X., Mihaylov, T., Mishra, P., Molybog, I., Nie, Y., Poulton, A., Reizenstein, J., Rungta, R., Saladi, K., Schelten, A., Silva, R., Smith, E. M., Subramanian, R., Tan, X. E., Tang, B., Taylor, R., Williams, A., Kuan, J. X., Xu, P., Yan, Z., Zarov, I., Zhang, Y., Fan, A., Kambadur, M., Narang, S., Rodriguez, A., Stojnic, R., Edunov, S., and Scialom, T. Llama 2: Open foundation and fine-tuned chat models, 2023. URL https://arxiv.org/abs/2307.09288.

Wang, S., Wei, J., Sabne, A., Davis, A., Ilbeyi, B., Hechtman, B., Chen, D., Murthy, K. S., Maggioni, M., Zhang, Q., Kumar, S., Guo, T., Xu, Y., and Zhou, Z. Overlap Communication with Dependent Computation via Decomposition in Large Deep Learning Models. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pp. 93–106, Vancouver BC Canada, December 2022. ACM. ISBN 978-1-4503-9915-9. doi: 10.1145/3567955.3567959. URL https://dl.acm.org/doi/10.1145/3567955.3567959.

Xu, Y., Lee, H., Chen, D., Hechtman, B., Huang, Y., Joshi, R., Krikun, M., Lepikhin, D., Ly, A., Maggioni, M., Pang, R., Shazeer, N., Wang, S., Wang, T., Wu, Y., and Chen, Z. GSPMD: General and Scalable Parallelization for ML Computation Graphs, December 2021. URL http://arxiv.org/abs/2105.04663.

Zheng, L., Li, Z., Zhang, H., Zhuang, Y., Chen, Z., Huang, Y., Wang, Y., Xu, Y., Zhuo, D., Xing, E. P., Gonzalez, J. E., and Stoica, I. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In Aguilera, M. K. and Weatherspoon, H. (eds.), *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pp. 559–578. USENIX Association, 2022. URL https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin.