DrJAX: Scalable and Differentiable MapReduce Primitives in JAX

Keith Rush [*1] **Zachary Charles** [*1] **Zachary Garrett** [1] **Sean Augenstein** [1] **Nicole Mitchell** [1]

Abstract

We present DrJAX, a JAX-based library designed to support large-scale distributed and parallel machine learning algorithms that use MapReduce-style operations. DrJAX leverages JAX's sharding mechanisms to enable native targeting of TPUs and state-of-the-art JAX runtimes, including Pathways (Barham et al., 2022). DrJAX embeds building blocks for MapReduce computations as primitives in JAX. This enables three key benefits. First, DrJAX computations can be translated directly to XLA HLO, enabling flexible integration with a wide array of ML training platforms. Second, DrJAX computations are fully differentiable. Last, DrJAX computations can be interpreted out to existing batch-processing compute systems, including traditional MapReduce systems like Apache Beam and cross-device compute systems like those powering federated learning applications. We show that DrJAX provides an easily programmable, performant, and scalable framework for parallelized algorithm development.

1. Introduction

The ability to scale abstractly written compute-intensive programs across large distributed compute environments is a key factor in the success of modern machine learning (ML). This is crucial for ML computations involving large language models, which are often too large to fit on a single compute node. Another key facet of modern ML software is the general ease with which computations can be written and optimized. Techniques such as automatic differentiation (AD) and just-in-time (JIT) compilation have enabled frameworks such as PyTorch (Paszke et al., 2019), TensorFlow (Abadi et al., 2016), and JAX (Bradbury et al., 2018) to scale to larger and more complex ML workloads.

[*]Equal contribution [1]Google Research. Correspondence to: Keith Rush <krush@google.com>, Zachary Charles <zachcharles@google.com>.

Accepted to the Workshop on Advancing Neural Network Training at International Conference on Machine Learning (WANT@ICML 2024).

These software frameworks generally focus on enabling parallelism for their most common use case: computation of a function's derivative across a batch of inputs. This computation is typically parallelized in two well-defined manners: across the batch dimension (i.e. data parallelism), and within the computation of the derivative at a single example (i.e. model parallelism). Data parallelism is extremely common across ML frameworks. Model parallelism is more complex, but has seen an explosion of progress in recent years (Gholami et al., 2018; Shazeer et al., 2018; Jia et al., 2019; Lepikhin et al., 2020; Xu et al., 2021) which has enabled the training and deployment of significantly larger models than previously possible (e.g. foundation models).

However, many sub-fields of ML do not fit neatly into this description, and instead employ parallelism over higher-level partitioned structures of data: in meta-learning (where data is partitioned across tasks) (Finn et al., 2017); in group-level differential privacy (where data is partitioned over discrete groups whose individual contributions to algorithm outputs are information-theoretically bounded) (Dwork, 2010); in model merging (Li et al., 2022) or "model soup" algorithms (where data, in the form of hyperparameters, is partitioned across model copies) (Li et al., 2022; Wortsman et al., 2022); in federated learning (where data is partitioned across clients who avoid directly sharing data) (McMahan et al., 2017); and in optimization with intermittent communication (where data is partitioned across model replicas) (Mangasarian and Solodov, 1993; Zinkevich et al., 2010; Zhang et al., 2016). The algorithms studied in these nominally different areas share many commonalities. An especially common factor is the use of the MapReduce (Dean and Ghemawat, 2004) programming paradigm, mapping parallel model training steps over partitioned data before invoking a reduction function. Data parallelism is a special case: we simply map and reduce over batches of data. In other applications, MapReduce is applied to coarser groups of data (e.g. multiple batches), which may additionally encode semantic structure of the underlying data.

While ML frameworks provide scalability, flexibility, and efficiency for data- and model-parallelism, an algorithm author who wishes to program over partitioned data in a parallel manner finds themselves in an awkward position. For example, frameworks for federated learning (e.g. Ingerman and Ostrowski (2019); Ziller et al. (2021); Ro et al. (2021);

He et al. (2020)) offer parallelism over clients, but either do not support other ML use cases discussed above, or do not focus on large-scale datacenter performance. Underlying ML frameworks often offer powerful parallelism primitives (e.g. `jax.shard_map`), but these often assume (as is the case for `jax.shard_map`) that the specification of all model shardings and physical resources is known to the code author (Shazeer et al., 2018). By contrast, in DrJAX we wish to abstract out MapReduce-style computations, allowing them to be defined in terms of model forwards and backwards passes (for example) that are already sharded. Finally, we note that bridging research and production often requires translating computations (e.g. from JAX) to production platforms.

An ideal authoring surface for ML algorithms using MapReduce operations provides several features simultaneously: performant and scalable datacenter performance; the ability to decouple logical partitioning of data (number of groups of data to parallelize over) from physical compute (number of compute nodes); easy and extensible algorithm expression; JIT compilation and AD; and the capacity to translate algorithms to alternative infrastructure.

Contributions. We present DrJAX (**D**ifferentiable Map**R**educe **JAX**) a software library that brings the benefits of modern large-scale machine learning software – sharding, easy-to-use JIT compilation, and AD – to MapReduce-style algorithms operating on partitioned data. DrJAX embeds a simple mapping and reduction programming model, by decomposing computations into *differentiable* building blocks (Section 2). We fully implement this programming model in JAX by embedding these building blocks via JAX's `Primitive` mechanism (Section 3). This allows DrJAX to use powerful features like sharding and JIT compilation to XLA. For example, DrJAX can shard computations over data partitions, model, and within-data partitions simultaneously across physical and logical meshes of devices. Because DrJAX is essentially a careful programming of parallel algorithms in XLA, DrJAX can leverage advances in distributed datacenter training like GSPMD (Xu et al., 2021) and Pathways (Barham et al., 2022). We showcase the scaling and runtime benefits of DrJAX across a suite of large language model training experiments (Section 4).

JAX's `Primitive` mechanism also enables forward- and reverse-mode differentiation which DrJAX leverages to provide full differentiability of its MapReduce-style programs. By implementing the AD framework of Rush et al. (2023), we ensure that the derivative of a DrJAX program is simply another DrJAX program. This allows DrJAX and its AD system to be interpreted out to other platforms for parallel machine learning, including systems with strong guarantees about data locality and privacy (Section 5).

Related work. DrJAX draws inspiration from three main areas of software. First, DrJAX directly utilizes the programming model of MapReduce (Dean and Ghemawat, 2004) and its evolution in Apache Beam (Foundation), highlighting the power of focusing on replications, mappings and reductions on parallelized collections of data.

Second, much of DrJAX's treatment of machine learning and automatic differentiation is influenced by modern ML frameworks such as PyTorch (Paszke et al., 2019) and TensorFlow (Abadi et al., 2016). DrJAX's design is directly based on the functional-first nature of JAX (Bradbury et al., 2018). Several libraries implemented in JAX leverage similar (though not identical) mechanisms for ensuring scalability, notably the Praxis library for neural network layers (Google).

Finally, DrJAX is inspired by frameworks for federated learning. Without intending to be exhaustive, examples of such frameworks include: PySyft (Ziller et al., 2021), FedJAX (Ro et al., 2021), FedScale (Lai et al., 2022), FedML (He et al., 2020), Flower (Beutel et al., 2020), FLUTE (Dimitriadis et al., 2022), FL_Pytorch (Burlachenko et al., 2021), and FATE (Liu et al., 2021). In particular, DrJAX's programming model was significantly influenced by TensorFlow Federated (Ingerman and Ostrowski, 2019) and its *federated computations* (Charles et al., 2022). Moreover, DrJAX uses the AD framework for federated computations proposed by Rush et al. (2023) to extend AD to MapReduce computations more broadly.

2. System Design

DrJAX is designed with two key ideas in mind. First, the types of parallel computing for machine learning discussed above can all be viewed as applications of MapReduce-style computations that use functions like model forward and backwards passes as black-box subroutines. Second, we can differentiate through MapReduce computations by using the AD techniques proposed by Rush et al. (2023). While their framework, federated AD, was motivated by federated learning, it can be directly adapted into a mechanism to apply AD to MapReduce computations. By combining standard AD techniques (e.g. computation graphs (Bauer, 1974)) with proper accounting of how data moves between logical partitions, we can express the derivative of a MapReduce computation as another MapReduce computation.

DrJAX operates on *partitioned* and *non-partitioned* values. A partition represents the data that we would like to perform MapReduce-style computations over. A non-partitioned value conceptually represents a singleton. We denote a non-partitioned value by v, and a partitioned value as $\mathbf{v} = [v_1, \ldots, v_n]$, where all v_i are elements of the same space. Note that the value of n depends on the partition, but the

ordering within the partition is arbitrary.

The inputs and outputs of DrJAX computations can include non-partitioned and partitioned values. Unlike classical MapReduce treatments, we do not assume that a computation will use a reducer. We consider a specific class of computations that can be built from the following computations, which we refer to as **building blocks**.

1. `broadcast`: Creates a partitioned value in which all groups have the same value (ie. $\texttt{broadcast}(v) = [v, v, \ldots, v]$).
2. `map_fn`: Applies a function f across a partition (ie. $\texttt{map\_fn}(f, \mathbf{v}) = [f(v_1), f(v_2), \ldots, f(v_n)]$).
3. `reduce_sum`: Sums over a partitioned value, returning a non-partitioned value (ie. $\texttt{reduce\_sum}(\mathbf{v}) = \sum_{i=1}^{n} v_i$).

This class is sufficiently expressive to include many parallel algorithms of interest, including parallel model-agnostic meta learning (MAML) (Finn et al., 2017), parallel and local SGD (Mangasarian and Solodov, 1993; Zinkevich et al., 2010), federated averaging (FedAvg) (McMahan et al., 2017), Branch-Train-Merge (Li et al., 2022), DiLoCo (Douillard et al., 2023), and many others.

For our purposes, the key fact about these computations is that they are *closed* under MapReduce AD. In particular, let $f : x \mapsto y$ where x is any collection of partitioned and non-partitioned inputs, and y is non-partitioned. If f can be composed from the building blocks above, then Rush et al. (2023) show that the function $\nabla f : x \mapsto {}^{dy}/_{dx}$ can also be expressed in terms of the building blocks above. Moreover, this can be done in a programmatic fashion. We will refer to this as *MapReduce AD* in the sequel.

This leads to our key observation: If we embed the building blocks above into JAX in a suitable manner, then we can (1) lower these computations to data structures accepted by performant data center runtimes, (2) implement MapReduce AD by appropriately delegating to JAX's AD, and (3) preserve partition information to enable translation to other production ML systems. DrJAX does just this, embedding the building blocks into JAX in a JIT-compatible manner. DrJAX also provides implementations of Jacobian-vector and vector-Jacobian products of the building blocks. This allows DrJAX to perform forward- and reverse-mode AD on MapReduce computations by delegating to JAX's forward- and reverse-mode AD.

Authoring surface. DrJAX code is almost entirely JAX code, with two general exceptions. First, there are the building blocks above. Second, DrJAX code must specify how many groups are in a partition during the invocation of the computation. To see this, consider the code in Snippet 1, which simply broadcasts a value across a partition, doubles the value in each group, and takes a sum over the partition.

```
import drjax

def broadcast_double_and_sum(x):
  y = drjax.broadcast(x)
  z = drjax.map_fn(lambda a: 2*a, y)
  return drjax.reduce_sum(z)
```

Snippet 1. An incomplete DrJAX program, which broadcasts x to a partition, doubles the value held by each group, and then sums over the partition. The program must know the partition size to correctly compute the desired result.

To compute the result, DrJAX needs to know the size of the partition. The user has to give this information to the DrJAX programs. For example, Snippet 2 modifies Snippet 1 to include explicit information about the partition size (ie. the number of groups in the partition).

```
@drjax.program(partition_size=3)
def broadcast_double_and_sum(x):
  y = drjax.broadcast(x)
  z = drjax.map_fn(lambda a: 2*a, y)
  return drjax.reduce_sum(z)
```

Snippet 2. A basic DrJAX program, with a decorator specifying the partition size. With this information, DrJAX can determine that the program should return $6x$.

3. Implementation

We now discuss DrJAX's implementation in JAX, in particular how it represents partitioned values and implements computations on them. We also discuss how we ensure DrJAX computations are effectively sharded across data center runtimes, and how DrJAX can implement MapReduce AD. While we focus on the programming model above, we note DrJAX's lower-level implementation can be used for much more general distributed and even hierarchical processing and sharding patterns.

Paritioned values. DrJAX represents both partitioned values as arrays with an extra leading dimension indicating the number of groups associated to them. Compared to partitioned values, non-partitioned values have an extra leading axis of cardinality equal to the number of groups. Given an $(d+1)$-dimensional array x, the i-th component $x[i, ...]$ is the d-dimensional array held by the i-th group. Figure 1 gives an example of this representation.

All JAX values are essentially represented as structures whose leaf nodes are arrays (referred to as pytrees in JAX), which DrJAX carries forward. A partitioned structure is a

structure (ie. a pytree) of partitioned arrays. For example, Figure 2 gives an example of a partitioned structure with multiple leaf arrays.

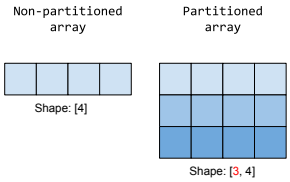

Figure 1. DrJAX's representation of a non-partitioned array (left) and an array partitioned over 3 groups (right).

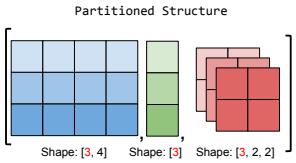

Figure 2. A partitioned structure in DrJAX with 3 groups. Each leaf is a partitioned array.

DrJAX computations. Since partitioned values are represented as JAX arrays, DrJAX computations must operate on JAX arrays. Other goals of DrJAX, like scalability, data center performance, and enabling differentiability, inform how DrJAX operates on arrays. We address these simultaneously by leveraging JAX's `Primitive` mechanism.

Briefly, DrJAX defines the building blocks above at decorator-installation time. These building blocks are processed *symbolically* by functional transformations in JAX. DrJAX registers the behavior of these operators under the action of these transformations, providing JAX with the necessary information to (1) lower DrJAX-defined functions wholesale to XLA HLO, (2) shard intermediate tensors in a maximally efficient manner, and (3) transform JAX functions containing DrJAX code under operations including JIT compilation and differentiation.

Given the representation of partitioned values above, we can implement the building blocks via straightforward array operations:

1. `broadcast`: Tile an array over its leading axis.
2. `map_fn`: Apply a function pointwise over an array's leading axis.
3. `reduce_sum`: Sum an array over its leading axis.

We extend these to partitioned structures by applying them leaf-wise. DrJAX registers these implementations with JAX lowering logic. This ensures that DrJAX code is entirely replaced by JAX code by the time JAX dispatches logic to an XLA runtime. Other building blocks can be added to DrJAX by registering primitives in a similar fashion, or by defining them in terms of the building blocks above. For example, DrJAX provides a `reduce_mean` symbol which takes an average across groups in a partitioned array, which lowers to two calls to `reduce_sum`.

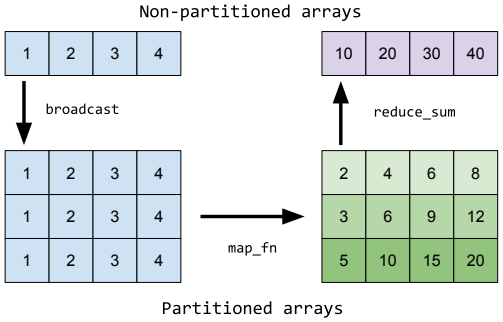

Figure 3. A high-level depiction of DrJAX building blocks operating on and transforming non-partitioned and partitioned arrays.

Sharding DrJAX computations. By registering the primitives above, we ensure that compilers like GSPMD (Xu et al., 2021) can shard DrJAX computations across worker nodes. Critically, and distinct from paradigms such as `jax.pmap`, DrJAX decouples partition size from sharding computations. A partition of size n is purely logical, and can be sharded across any number of m workers as long as $m|n$. We want to ensure that, no matter how many workers we shard over, DrJAX computations are as efficient as possible. To do so, we only need to focus on how the building blocks above are sharded by compilers. Once this is done, we are free to compose with model- and data-parallelism provided by various JAX libraries.

While some DrJAX building blocks are trivially parallelizable (e.g. `map_fn`), compilers may not be able to detect this and generate efficient code. As noted by Xu et al. (2021) and Lepikhin et al. (2020), internal sharding annotations can dramatically affect the performance of a compiler targeting distributed execution. DrJAX uses static and dynamic sharding constraints to ensure that after compilation, the resulting computation will run efficiently in the data center. As we will see in Section 4, without these annotations, compilers like GSPMD do not optimally shard DrJAX computations, especially as the partition size increases.

Derivatives of DrJAX computations. The last benefit of embedding building blocks as JAX primitives is that it gives us a straightforward way to take derivatives of DrJAX computations using AD. We refer to this as *MapReduce AD*. To do so, we only need to define the action of vector-Jacobian products (VJPs) and Jacobian-vector products (JVPs) on the DrJAX primitives. Rush et al. (2023) discuss how to compute these products, and show that their computation does not require any new building blocks. That is, the JVPs and VJPs of these primitives can be expressed in terms of the same set of primitives. With the JVPs and VJPs, we can now entirely rely on JAX's AD to do forward- and reverse-mode AD on computations involving these primitives. For

more details, see Section 5.

4. Scalability and Efficiency

We now present numerical evidence of the scalability and efficiency of DrJAX, by testing DrJAX in a distributed training setting. We perform multiple rounds of local SGD (Zhang et al., 2016) on transformer language models with 350 million (350M), 1 billion (1B), and 8 billion (8B) parameters. We use a causal language modeling loss and a sequence length of 512. In every round, we parallelize 4 local SGD steps, each with batch size 8, across some number of data groups from a partition before synchronization. We use a partitioned version of CCNews dataset, where news articles are partitioned according to their base URL domain. We use Dataset Grouper (Charles et al., 2023) to iterate over groups of data efficiently. In each round, we sample some number of data groups, which form a partition of some size. We vary the partition size proportionally to the number of workers used in the computation. To describe the scale of the experiments, Table 1 contains the maximum number of tokens processed and model parameters updated per round for each model.

For all experiments, we shard the training computation over some number of TPUv2s. The total number of TPU chips, m, is proportional to the partition size, n. For 350M, 1B, and 8B models we use n, $4n$, and $8n$ chips in total, respectively. This means that if we double the partition size, we also double the number of TPU chips used. We fully shard the computations across the workers, and additionally do model parallelism for the 1B and 8B models. For all experiments, our DrJAX computations are first compiled using GSPMD (Xu et al., 2021) and then delegated to the Pathways runtime (Barham et al., 2022).

Weak scaling. The *weak scaling* of a system refers to how its compute time changes as the workload and compute resources scale simultaneously. Generally, modern ML systems attempt to obtain near-constant weak scaling performance.[1] For DrJAX, we fix the model size and number of local SGD steps computed per data group in the partition, and vary the partition size and number of workers proportionally in order to vary the workload size. As discussed above, we scale the number of TPU chips used in our simulations linearly with respect to the partition size.

Figure 4 shows how training time of DrJAX-based local SGD scales as the partition size and number of TPU chips increase, across a range of model sizes. DrJAX-exhibits near-constant runtime for a fixed model size, even up to a pool of 128 or 512 workers. This is highly non-trivial.

[1]Constant performance is generally impossible due to overhead such as synchronization costs.

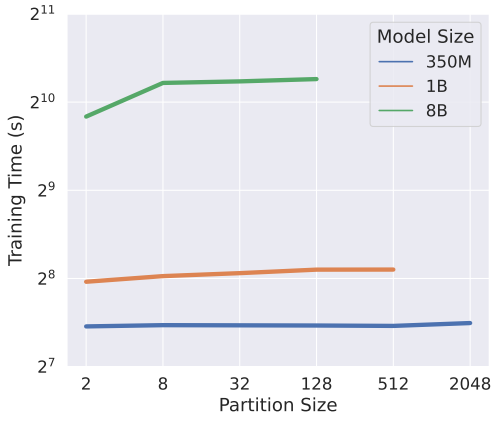

Figure 4. Total training time for 100 rounds of local SGD on various transformer language models sizes, with varying partition sizes.

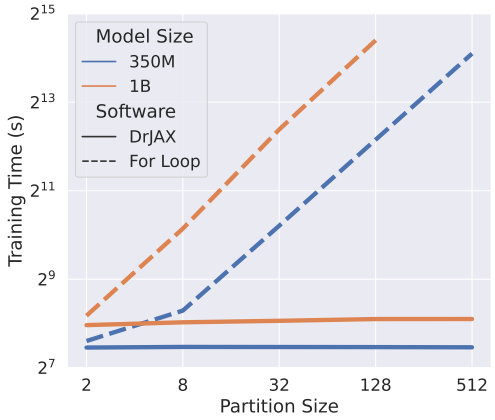

Figure 5. Total training time for 100 rounds of local SGD, with varying partition sizes. We implement local SGD using DrJAX and a python for-loop which we JIT compile.

Because local SGD involves parallel model training across workers, and for multiple steps per worker, the per-round workload size (in terms of total floating point operations) is at least as large as 4×(model size)×(partition size). To see this, note that in each round, for each group in the data partition, we update a local model copy 4 times. As shown in Table 1, the largest workload for each model size involves updating over 1 trillion model parameters per round.

JIT compilation alone is not enough. ML research often involves writing custom training loops. A naive implementation of local SGD, often used for research in distributed training, is simply a double for loop that iterates over workers in the pool, and over the batches held by each worker. The outer loop here has no data dependency, meaning that the values returned by iterations of this loop are not processed as inputs to the next iteration. One might therefore imagine

Table 1. Maximum partition size, partition size, number of workers, number of tokens processed, and total floating point operations (FLOPs) when training with local SGD, for each model size. For simplicity, we only present FLOPs associated with the forward pass, using the approximation that a forward pass on a model of size d uses d FLOPs.

Model Size	Partition Size	Num Workers	Tokens per Round	FLOPs per Round
350M	2048	2048	3.355×10^7	2.293×10^{13}
1B	512	512	8.389×10^6	1.638×10^{13}
8B	128	128	2.097×10^6	3.277×10^{13}

that a sufficiently advanced compiler could detect this fact, and parallelize worker training when possible (e.g. within resource constraints in the data center environment).

This can be a difficult task for a compiler. To illustrate this difficulty, we implemented a double for loop in place of DrJAX-based training (looping over workers, and over each worker's data). For both programs, we JIT-compiled the program, and provided identical input and output shardings to GSPMD and the XLA compiler stack. Though this stack is quite advanced and used to train many of the largest contemporary ML models, it does not recover the performance of DrJAX from this for-loop implementation. Indeed, round runtime scales linearly with the partition size (and therefore, the number of workers), as expected, rather than remaining constant, indicating an inability to use the increased resource scale allocated to the experiment.

GSPMD alone is not enough. A better way to parallelize across workers than the for-loop approach above is to implement DrJAX's MapReduce building blocks and use a compiler like GSPMD (Xu et al., 2021) to do automated sharding of the program. This leads to the question: Do we need DrJAX's internal sharding annotations to obtain weak scaling behavior, or can GSPMD alone fully and efficiently parallelize DrJAX computations? Given the relatively simple nature of local SGD's parallel processing patterns (heavily parallelized model training with infrequent synchronization), one might expect that isolating MapReduce building blocks as primitives with specially-designed sharding annotations is unnecessarily complex.

To test this, we took a DrJAX-based implementation of local SGD and removed all of DrJAX's internal sharding annotations at function-tracing time, denoting this DrJAX-NS (DrJAX with no sharding). We then re-ran the simulations in Figure 4. The results in Figure 6 show that at present, these explicit sharding annotations play a crucial role in ensuring DrJAX's weak-scaling behavior. DrJAX-NS computation times increased sublinearly but significantly faster than DrJAX computation times. Moreover, DrJAX-NS exhibited memory footprint scaling issues. We found that for sufficiently large model or worker pool sizes, DrJAX-NS eventually ran out of high-bandwidth memory. In particular, this occurred for the 1B model with 512 workers and for the 8B model with all tested numbers of workers; that is, at 2 workers and beyond.

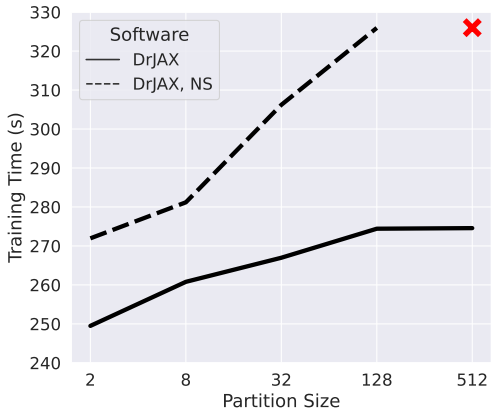

Figure 6. Total training time for 100 rounds of local SGD for the 1B model, with varying partition sizes. We implement local SGD using DrJAX with and without (DrJAX-NS) DrJAX's sharding annotations. The red X represents the point at which the DrJAX-NS could not be sharded without triggering out of memory errors.

5. Interpreting DrJAX to Other Platforms

While data center performance is the primary goal of DrJAX, we wish to preserve the optionality to translate DrJAX computations into artifacts interpretable by other systems, such as federated learning systems (Bonawitz et al., 2019; Paulik et al., 2021; Huba et al., 2022). For example, if `reduce_sum` is only captured as a `jnp.sum`, then it may be difficult to tell whether this sum is intended to be *within* a partition group, or across groups in a partition. Below, we discuss how DrJAX's implementation enables computing program representations that can be automatically translated to other platforms.

Preserving partition information. Recall from above that we implement MapReduce building blocks as JAX primitives, and build DrJAX computations out of these primitives. This has a key benefit when interpreting out to other systems: the ability to preserve information about how DrJAX building blocks are applied to partitioned data, which can inform things like cross-node communication

and computation boundaries within a production system.

JAX's `Primitive` mechanism allows users to inject new symbols into JAX itself, defining how these symbols behave under JAX's functional transformations like `jax.vmap` and `jax.grad`. These primitives are preserved in JAX's intermediate data structure, the `jaxpr`, which is usually later lowered into XLA HLO. By using a custom interpreter, we can instead generate and consume `jaxprs`. This custom interpreter can use special behavior when it encounters the `DrJAX`-defined symbols injected via the `Primitive` mechanism. This preserves information about data partitioning and operations within and across partitions, allowing us to translate `jaxprs` into computations that can be run by other platforms.

An example `jaxpr` is illustrative. In Snippet 3, we define a `DrJAX` program for computing the MAML loss from Finn et al. (2017). This is the loss of a model on some data *after* some number of steps of (stochastic) gradient descent. For our purposes, we do a single gradient descent step before evaluating the loss, using some user-specified learning rate. In the nomenclature of meta-learning, the data is partitioned across some number of *tasks*, and we assume we have access to some loss function $\texttt{loss}(x, y)$ where x is the model, and y is the task.

```
def maml_loss(model, lr, task):
  g = jax.grad(loss)(model, task)
  model = model - lr * g
  return loss(model, task)
```

Snippet 3. MAML loss

This loss is easily parallelized across workers using `DrJAX`, as in Snippet 4. The algorithm is straightforward: The model and learning rate are broadcast to every task, the MAML loss is computed using the model, learning rate, and task, and the resulting loss values are averaged.

```
@drjax.program(partition_size=3)
def parallel_maml_loss(model, lr, tasks):
  model = drjax.broadcast(model)
  lr = drjax.broadcast(lr)
  losses = drjax.map_fn(
    maml_loss, (model, lr, tasks))
  return drjax.reduce_mean(client_losses)
```

Snippet 4. Computing the average meta-learning loss over a data partition of size 3 via `DrJAX`.

To obtain a `jaxpr` representing this processing pattern, we provide the concrete shape and type of arguments. For brevity, we assume the model and tasks are scalars, and the loss function is the square loss (ie. $\texttt{loss}(x, y) = (x - y)^2$).

Given this information, JAX can generate a `jaxpr` representing Snippet 4. The result is in Snippet 5. The key takeaway is that this `jaxpr` preserves the `DrJAX`-defined primitives representing cross-machine communication, **`broadcast`** and **`reduce_mean`**, both of which are primitives registered by `DrJAX` in JAX. We can trace through the arguments in the `jaxpr` to see that the computation operates by (1) broadcasting values (the model and learning rate) across the partition, (2) calculating `loss` using the broadcast values, (3) taking a mean over the partition.

```
{ lambda ; a:f32[] b:f32[] c:f32[3]. let
    d:f32[1] = broadcast a
    e:f32[1] = broadcast b
    f:f32[1] = sub d e
    _:f32[1] = integer_pow[y=2] f
    g:f32[1] = integer_pow[y=1] f
    h:f32[1] = mul 2.0 g
    i:f32[1] = mul 1.0 h
    j:f32[1] = mul c i
    k:f32[1] = sub d j
    l:f32[1] = sub k e
    m:f32[1] = integer_pow[y=2] l
    n:f32[] = reduce_mean m
  in (n,) }
```

Snippet 5. `jaxpr` generated for `parallel_maml_loss`.

Interpreting the `jaxpr` to a production systems, especially distributed systems, such as TensorFlow Federated (Ingerman and Ostrowski, 2019) is now straightforward: all cross-machine communication is explicit, and the processing in-between communication is entirely local and can be extracted into standalone functions executed locally by compute nodes in the system.

Integrating MapReduce AD. As discussed in Section 2, the `Primitive` mechanism allows `DrJAX` to to specify the behavior of building blocks under JAX's functional transformations, including computing forward- and reverse-mode Jacobians (`jax.jacfwd` and `jax.jacrev`). This allows `DrJAX` to apply AD to MapReduce computations via the forward- and reverse-mode algorithms presented in Rush et al. (2023). For example, forward- and reverse-mode Jacobians of `broadcast` can be computed via `broadcast` and `reduce_sum`, respectively. `DrJAX` can therefore implement MapReduce AD without additional primitives. This means that the `jaxpr` of `DrJAX` computations that use MapReduce AD will contain JAX's standard AD symbols, along with `DrJAX`'s primitive set, ensuring that computations using MapReduce AD are still interpretable to other systems.

For example, Snippet 6 gives the `jaxpr` of the reverse-mode gradient of `parallel_maml_loss)` (ie. the deriva-

tive of the parallel MAML loss computation in Snippet 4). Again, we see that information about communication in the system is preserved. The `jaxpr` contains the primitives **`broadcast`**, **`reduce_mean`**, and **`reduce_sum`**, and which just as above, can be used by a custom interpreter to translate the `jaxpr` into a production system.

```
{ lambda ; a:f32[] b:f32[] c:f32[3]. let
    d:f32[1] = broadcast a
    e:f32[1] = broadcast b
    f:f32[1] = sub d e
    _:f32[1] = integer_pow[y=2] f
    g:f32[1] = integer_pow[y=1] f
    _:f32[1] = mul 2.0 g
    h:f32[1] = integer_pow[y=1] f
    i:f32[1] = integer_pow[y=0] f
    j:f32[1] = mul 1.0 i
    k:f32[1] = mul 2.0 h
    l:f32[1] = mul 1.0 k
    m:f32[1] = mul c l
    n:f32[1] = sub d m
    o:f32[1] = sub n e
    p:f32[1] = integer_pow[y=2] o
    q:f32[1] = integer_pow[y=1] o
    r:f32[1] = mul 2.0 q
    _:f32[] = reduce_mean p
    s:f32[1] = broadcast 1.0
    t:f32[1] = div s 1.0
    u:f32[1] = mul t r
    v:f32[1] = neg u
    w:f32[1] = mul c v
    x:f32[1] = mul 1.0 w
    y:f32[1] = mul 2.0 x
    z:f32[1] = mul y j
    ba:f32[1] = add_any u z
    bb:f32[] = reduce_sum ba
  in (bb,) }
```

Snippet 6. `jaxpr` generated for `jax.grad(parallel_maml_loss)`.

6. Discussion

Why MapReduce AD? While features like scalability and efficiency are self-explanatory, the reader may be interested in *why* we wish to implement MapReduce AD, especially given the care required to interpret to production systems. In short, MapReduce AD makes expressing efficient algorithms easier (Rush et al., 2023). By way of analogy, AD has made the development of sophisticated neural network architectures significantly easier. Libraries can define the conceptually simpler forward-pass, and rely on AD to perform backpropagation. The result is often faster and less error-prone than hand-implemented gradient computations (Baydin et al., 2018).

Algorithms that operate on partitioned data can see similar benefits. For example, Snippet 4 contains a `DrJAX` program used to compute the average MAML loss over tasks, as in meta-learning. By simply calling `jax.grad(parallel_maml_loss)`, we immediately get a `DrJAX` program that computes the average MAML gradient over tasks. Using this, we can easily write an algorithm that efficiently parallelizes the MAML algorithm. Snippet 7 depicts this, defining implementing a parallel MAML algorithm by simply pairing `jax.grad` with an SGD update step.

```
@drjax.program(partition_size=3)
def parallel_maml_step(model, lr, tasks):
  g = jax.grad(parallel_maml_loss)(model,
     lr, tasks)
  return model - lr * g
```

Snippet 7. Implementing Parallel MAML via MapReduce AD.

Self-tuning algorithms. Another potential use case for MapReduce AD is creating self-tuning algorithms, which use AD to optimize algorithmic hyperparameters (e.g. *hypergradient descent*). By using MapReduce AD, we can automatically adjust hyperparameters that govern underlying optimization algorithms, but also adjust the MapReduce operations themselves. For example, Rush et al. (2023) show that the weights governing a weighted mean-based reduction can be learned in tandem with performing federated learning. Similarly, Wang et al. (2023) derive formulas for hypergradient descent on federated learning algorithms, but this process can be slow, error-prone, and algorithm-specific. By contrast, Rush et al. (2023) use an AD system (that inspired MapReduce AD, as we discuss in Section 3)) to do the same, without needing to derive algorithm-specific rules. More generally, MapReduce AD opens the door to a wide variety of self-tuning distributed and parallel algorithms.

Conclusion. By pairing differentiable MapReduce primitives with an easy-to-use front-end via JAX, performant building block implementations, and useful sharding information, we hope to accelerate research on distributed and parallel ML. Future work includes (1) generalizations of `DrJAX` to non-linear reductions, (2) extensions of `DrJAX` to more general types of data, including hierarchically partitioned data, and (3) mature `DrJAX` interpreters for specific production systems.

References

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner,

Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 265–283. USENIX Association, 2016. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Dan Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, Brennan Saeta, Parker Schuh, Ryan Sepassi, Laurent El Shafey, Chandramohan A. Thekkath, and Yonghui Wu. Pathways: Asynchronous distributed dataflow for ML. In Diana Marculescu, Yuejie Chi, and Carole-Jean Wu, editors, *Proceedings of Machine Learning and Systems 2022, MLSys 2022, Santa Clara, CA, USA, August 29 - September 1, 2022*. mlsys.org, 2022. URL https://proceedings.mlsys.org/paper/2022/hash/98dce83da57b0395e163467c9dae521b-Abstract.html.

Friedrich L Bauer. Computational graphs and rounding error. *SIAM Journal on Numerical Analysis*, 11(1):87–96, 1974.

Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research*, 18(153):1–43, 2018. URL http://jmlr.org/papers/v18/17-468.html.

Daniel J. Beutel, Taner Topal, Akhil Mathur, Xinchi Qiu, Titouan Parcollet, and Nicholas D. Lane. Flower: A friendly federated learning research framework. *CoRR*, abs/2007.14390, 2020. URL https://arxiv.org/abs/2007.14390.

Kallista A. Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloé Kiddon, Jakub Konečný, Stefano Mazzocchi, Brendan McMahan, Timon Van Overveldt, David Petrou, Daniel Ramage, and Jason Roselander. Towards federated learning at scale: System design. In Ameet Talwalkar, Virginia Smith, and Matei Zaharia, editors, *Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, March 31 - April 2, 2019*. mlsys.org, 2019. URL https://proceedings.mlsys.org/book/271.pdf.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

Konstantin Burlachenko, Samuel Horváth, and Peter Richtárik. FL_PyTorch: Optimization research simulator for federated learning. In *Proceedings of the 2nd ACM International Workshop on Distributed Machine Learning*, pages 1–7, 2021.

Zachary Charles, Kallista Bonawitz, Stanislav Chiknavaryan, Brendan McMahan, et al. Federated select: A primitive for communication-and memory-efficient federated learning. *arXiv preprint arXiv:2208.09432*, 2022.

Zachary Charles, Nicole Mitchell, Krishna Pillutla, Michael Reneer, and Zachary Garrett. Towards federated foundation models: Scalable dataset pipelines for group-structured learning. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/662bb9c4dcc96aeaac8e7cd3fc6a0add-Abstract-Datasets_and_Benchmarks.html.

Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

Dimitrios Dimitriadis, Mirian Hipolito Garcia, Daniel Madrigal Diaz, Andre Manoel, and Robert Sim. FLUTE: A scalable, extensible framework for high-performance federated learning simulations. *CoRR*, abs/2203.13789, 2022. doi: 10.48550/ARXIV.2203.13789. URL https://doi.org/10.48550/arXiv.2203.13789.

Arthur Douillard, Qixuang Feng, Andrei A. Rusu, Rachita Chhaparia, Yani Donchev, Adhiguna Kuncoro, Marc'Aurelio Ranzato, Arthur Szlam, and Jiajun Shen. Diloco: Distributed low-communication training of language models. *CoRR*, abs/2311.08105, 2023. doi: 10.48550/ARXIV.2311.08105. URL https://doi.org/10.48550/arXiv.2311.08105.

Cynthia Dwork. Differential privacy in new settings. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 174–183. SIAM, 2010.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International conference on machine learning*, pages 1126–1135. PMLR, 2017.

Apache Software Foundation. Apache beam. URL https://beam.apache.org/.

Amir Gholami, Ariful Azad, Peter Jin, Kurt Keutzer, and Aydin Buluc. Integrated model, batch, and domain parallelism in training neural networks. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 77–86, 2018.

LLC Google. Praxis. URL https://github.com/google/praxis.

Chaoyang He, Songze Li, Jinhyun So, Mi Zhang, Hongyi Wang, Xiaoyang Wang, Praneeth Vepakomma, Abhishek Singh, Hang Qiu, Li Shen, Peilin Zhao, Yan Kang, Yang Liu, Ramesh Raskar, Qiang Yang, Murali Annavaram, and Salman Avestimehr. Fedml: A research library and benchmark for federated machine learning, 07 2020.

Dzmitry Huba, John Nguyen, Kshitiz Malik, Ruiyu Zhu, Mike Rabbat, Ashkan Yousefpour, Carole-Jean Wu, Hongyuan Zhan, Pavel Ustinov, Harish Srinivas, Kaikai Wang, Anthony Shoumikhin, Jesik Min, and Mani Malek. Papaya: Practical, private, and scalable federated learning. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 814–832, 2022. URL https://proceedings.mlsys.org/paper_files/paper/2022/file/a8bc4cb14a20f20d1f96188bd61eec87-Paper.pdf.

Alex Ingerman and Krzysztof Ostrowski. Introducing Tensorflow Federated, Mar 2019. URL https://blog.tensorflow.org/2019/03/introducing-tensorflow-federated.html.

Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *Proceedings of Machine Learning and Systems*, 1:1–13, 2019.

Fan Lai, Yinwei Dai, Sanjay S. Singapuram, Jiachen Liu, Xiangfeng Zhu, Harsha V. Madhyastha, and Mosharaf Chowdhury. FedScale: Benchmarking model and system performance of federated learning at scale. In *International Conference on Machine Learning (ICML)*, 2022.

Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. GShard: Scaling giant models with conditional computation and automatic sharding. *CoRR*, abs/2006.16668, 2020. URL https://arxiv.org/abs/2006.16668.

Margaret Li, Suchin Gururangan, Tim Dettmers, Mike Lewis, Tim Althoff, Noah A Smith, and Luke Zettlemoyer. Branch-train-merge: Embarrassingly parallel training of expert language models. *arXiv preprint arXiv:2208.03306*, 2022.

Yang Liu, Tao Fan, Tianjian Chen, Qian Xu, and Qiang Yang. FATE: an industrial grade platform for collaborative learning with data protection. *Journal of Machine Learning Research*, 22(226):1–6, 2021. URL http://jmlr.org/papers/v22/20-815.html.

O. L. Mangasarian and M. V. Solodov. Backpropagation convergence via deterministic nonmonotone perturbed minimization. In J. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems*, volume 6. Morgan-Kaufmann, 1993. URL https://proceedings.neurips.cc/paper_files/paper/1993/file/4558dbb6f6f8bb2e16d03b85bde76e2c-Paper.pdf.

Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-efficient learning of deep networks from decentralized data. In Aarti Singh and Xiaojin (Jerry) Zhu, editors, *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, AISTATS 2017, 20-22 April 2017, Fort Lauderdale, FL, USA*, volume 54 of *Proceedings of Machine Learning Research*, pages 1273–1282. PMLR, 2017. URL http://proceedings.mlr.press/v54/mcmahan17a.html.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 8024–8035, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

Matthias Paulik, Matt Seigel, Henry Mason, Dominic Telaar, Joris Kluivers, Rogier C. van Dalen, Chi Wai Lau, Luke Carlson, Filip Granqvist, Chris Vandevelde, Sudeep Agarwal, Julien Freudiger, Andrew Byde, Abhishek Bhowmick, Gaurav Kapoor, Si Beaumont, Áine Cahill, Dominic Hughes, Omid Javidbakht, Fei Dong, Rehan Rishi, and Stanley Hung. Federated evaluation and tuning for on-device personalization: System design & applications. *CoRR*, abs/2102.08503, 2021. URL https://arxiv.org/abs/2102.08503.

Jae Hun Ro, Ananda Theertha Suresh, and Ke Wu. FedJAX: Federated learning simulation with JAX. *arXiv preprint arXiv:2108.02117*, 2021.

Keith Rush, Zachary Charles, and Zachary Garrett. Federated automatic differentiation. *arXiv preprint arXiv:2301.07806*, 2023.

Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. Mesh-tensorflow: Deep learning for supercomputers. *Advances in neural information processing systems*, 31, 2018.

Ziyao Wang, Jianyu Wang, and Ang Li. Fedhyper: A universal and robust learning rate scheduler for federated learning with hypergradient descent. In *The Twelfth International Conference on Learning Representations*, 2023.

Mitchell Wortsman, Gabriel Ilharco, Samir Ya Gadre, Rebecca Roelofs, Raphael Gontijo-Lopes, Ari S Morcos, Hongseok Namkoong, Ali Farhadi, Yair Carmon, Simon Kornblith, et al. Model soups: averaging weights of multiple fine-tuned models improves accuracy without increasing inference time. In *International conference on machine learning*, pages 23965–23998. PMLR, 2022.

Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake A. Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, Ruoming Pang, Noam Shazeer, Shibo Wang, Tao Wang, Yonghui Wu, and Zhifeng Chen. GSPMD: general and scalable parallelization for ML computation graphs. *CoRR*, abs/2105.04663, 2021. URL https://arxiv.org/abs/2105.04663.

Jian Zhang, Christopher De Sa, Ioannis Mitliagkas, and Christopher Ré. Parallel sgd: When does averaging help? *arXiv preprint arXiv:1606.07365*, 2016.

Alexander Ziller, Andrew Trask, Antonio Lopardo, Benjamin Szymkow, Bobby Wagner, Emma Bluemke, Jean-Mickael Nounahon, Jonathan Passerat-Palmbach, Kritika Prakash, Nick Rose, et al. Pysyft: A library for easy federated learning. *Federated learning systems: Towards next-generation AI*, pages 111–139, 2021.

Martin Zinkevich, Markus Weimer, Lihong Li, and Alex Smola. Parallelized stochastic gradient descent. *Advances in neural information processing systems*, 23, 2010.