DistIR: An Intermediate Representation for Optimizing Distributed Neural Networks

Keshav Santhanam
keshav2@stanford.edu
Stanford University, USA

Siddharth Krishna
Ryota Tomioka
Andrew Fitzgibbon
Tim Harris
{t-sikris,ryoto,awf,tiharr}@microsoft.com
Microsoft, UK

Abstract

The rapidly growing size of deep neural network (DNN) models and datasets has given rise to a variety of distribution strategies such as data, horizontal, and pipeline parallelism. However, selecting the best set of strategies for a given model and hardware configuration is challenging because debugging and testing on clusters is expensive. In this work we propose DistIR, an IR for explicitly representing distributed DNN computation that can capture many popular distribution strategies. We build an analysis framework for DistIR programs, including a simulator and reference executor that can be used to automatically search for an optimal distribution strategy. Our unified global representation also eases development of new distribution strategies, as one can reuse the lowering to per-rank backend programs. Preliminary results using a grid search over a hybrid data/horizontal/pipeline-parallel space suggest DistIR and its simulator can aid automatic DNN distribution.

CCS Concepts: • **Computing methodologies** → **Distributed programming languages**; **Machine learning**; • **Software and its engineering** → **Compilers**.

Keywords: intermediate representation, distributed computation, neural networks, optimization

ACM Reference Format:
Keshav Santhanam, Siddharth Krishna, Ryota Tomioka, Andrew Fitzgibbon, and Tim Harris. 2021. DistIR: An Intermediate Representation for Optimizing Distributed Neural Networks. In *The 1st Workshop on Machine Learning and Systems (EuroMLSys '21), April 26, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3437984.3458829

EuroMLSys '21, April 26, 2021, Online, United Kingdom
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8298-4/21/04…$15.00
https://doi.org/10.1145/3437984.3458829

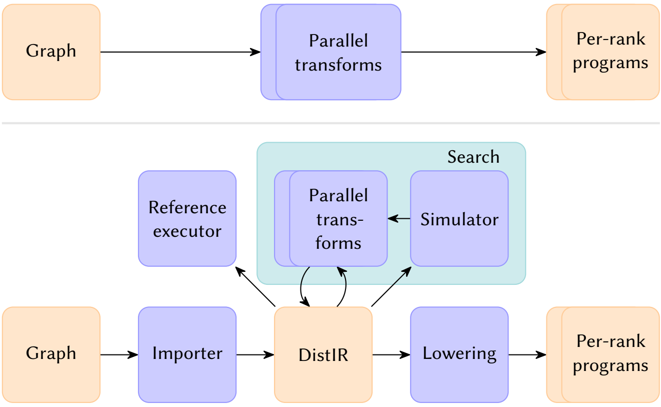

Figure 1. The workflow of existing approaches to DNN distribution (top) vs. DistIR's approach (bottom). Instead of directly transforming the computation graph to per-rank programs, we write parallel transforms as manipulating DistIR programs, which can then be lowered for execution.

1 Introduction

In recent years, deep neural network (DNN) models and datasets have grown rapidly in size [5, 10, 21, 29]. Large datasets such as C4 [27] are impractical to process on a single device, and models with billions of parameters [10] exceed the memory capacity of even the highest grade accelerators. Thus, it is often important to distribute models across multiple devices.

DNN distribution strategies each make different trade-offs to tailor for particular model architectures or hardware types. For instance, data parallelism [9] partitions input data across devices or ranks, allowing one to train models with large batch sizes, but can incur high communication costs to sync multiple copies of the model's parameters. Other distribution strategies, such as horizontal model parallelism [31], pipeline parallelism [14, 24], and hybrid strategies that combine parallelism dimensions [17, 18, 21, 25], allow one to scale model size but have their own drawbacks; e.g., pipeline parallelism needs a large batch size to keep all devices utilized. In this work we address the following questions:

> How do we select the best distribution strategy for a given model and hardware configuration?

> How do we develop new strategies for emerging model architectures?

Unfortunately, most existing DNN frameworks suffer from one or more of the following problems:

1. Choosing the best strategy for a given model, or designing a new strategy for a novel architecture, is difficult because testing and debugging is expensive on real clusters. Existing frameworks do not capture the distribution strategy explicitly in an intermediate representation (IR) [24, 30, 31] (Figure 1, top), which makes it hard to perform static analyses such as simulation or runtime/memory estimation. Such analyses both accelerate the task of manual distribution and are vital for systems that automatically distribute models [17, 33].
2. Combining existing distribution strategies is challenging because many implementations [14, 21, 24, 31] are not written as composable transforms; in fact, they often have different input and output representations.
3. Implementing a new strategy in existing frameworks is difficult as this entails designing the distribution algorithm while working with low-level programs; these frameworks provide no way to decouple the distribution from the creation of the per-rank programs.

In this work we propose a more modular workflow for distributing DNNs (Figure 1, bottom) using DistIR, a new IR for distributed computation. Our key insight is that all three of these problems can be solved by capturing the entire distributed execution in an IR and implementing distribution strategies as transformations over this IR.

Capturing distribution explicitly in the IR design enables efficient simulation and analysis. Problem 1 can be tackled by using a simulator that estimates runtime and memory consumption of DistIR programs in an automated search algorithm to pick the best distribution for a given model. We present an abstract interpretation framework [7, 8] that can be instantiated to perform a variety of analyses such as simulation in linear time. Our framework can also be used to construct a reference executor that executes a DistIR program on a single device. These tools can help performance engineers quickly test and debug new distribution strategies.

A uniform representation also makes it possible to write hybrid strategies as compositions of IR transformations (Problem 2). Implementing a new strategy, Problem 3, is simplified as one can focus on achieving the desired distribution in a high-level IR and reuse the lowering pass that produces the low-level per-rank programs. An IR can also enable high-level (e.g. Python) APIs for DNN practitioners to directly write and combine distribution strategies.

The key properties of DistIR are its explicit *schedule* and *distributed semantics*. DistIR programs consist of a sequence of operations that are executed in the order that they appear. DistIR's semantics are defined over a distributed computing model where each device is capable of executing one operation at a time, and operations involving multiple devices execute synchronously on all participating devices (e.g., `Send` blocks both sender and receiver). Despite this apparent restriction, we show that DistIR can express the range of currently employed distribution strategies in deep learning.

DistIR's benefits are not due to special primitives, but to its explicit modeling of the global distributed computation. DistIR uses MLIR's dialect-based design [19] to be parametric with respect to its primitive operators. This allows one to easily extend DistIR to new application domains by providing operator definitions and cost models. In this paper, we instantiate DistIR with the ONNX operators for machine learning primitives and the MPI communication primitives, and use analytic cost models. One could instead instantiate it with the primitive operators of XLA or PyTorch in order to integrate it with their respective frameworks.

In summary, this paper makes the following contributions:

- We present DistIR, an efficient and expressive IR for distributed computation (§2). DistIR can express many popular distribution strategies, and DistIR's modular design allows adding new primitives to capture novel strategies.
- We define an analysis framework (§3), that can be used for efficient lowering of DistIR (§3.1), a reference executor to check correctness of DistIR transforms (§3.2), and a fast general-purpose simulator for estimating runtime and memory consumption of DistIR programs (§3.3).
- We show how DistIR facilitates writing parallel transforms by implementing a prototype transform for the D/H/P search space [30] created by combining data, horizontal (or operator), and pipeline parallelism.
- We demonstrate DistIR's suitability for automatic distribution and optimization by applying a grid search algorithm over the D/H/P space (§4).

We discuss future work in §5, review related work in §6, and conclude in §7.

2 DistIR

In this section we define the DistIR language and semantics, and discuss its design and expressivity.

DistIR is an intermediate representation (IR) for distributed computation based on the Multi-level Intermediate Representation (MLIR) [19] compiler framework. As such, its syntax is based on the Static Single Assignment (SSA) form. While DistIR is designed to be implemented within the MLIR framework, for ease of presentation, in this paper we use a simplified syntax (e.g., Figure 2).

The top-level container is a module, which is comprised of a sequence of functions. A function consists of a name, a sequence of variables that are function parameters, and a sequence of operations that make up the function body.[1] Operations come in three forms: invocations to a primitive

[1] We use MLIR naming conventions in code listings, e.g. `%x` for variables and `@foo` for functions.

```
 1 func @dense(%w, %x) {
 2   %A, %b = UnpackTuple(%w)
 3   %h = Gemm(%x, %A, %b)
 4   %a = Relu(%h)
 5   return %a
 6 }
 7
 8 func @denseGrad(%wb, %x, %da) { ... }
 9
10 func @lossGrad(%p, %y) { ... }
11
12 func @mlpPP(%w1: D1, %w2: D2, %x: D1, %y: D2) {
13   // Split into microbatches 1 and 2
14   %x_1: D1, %x_2: D1 = Split(%x, dim=0, num_splits=2)
15   %y_1: D2, %y_2: D2 = Split(%y, dim=0, num_splits=2)
16   // Pipeline
17   %as_1: D1 = @dense(%w1, %x_1)
18   %ar_1: D2 = Send(%as_1, 2)
19   %as_2: D1 = @dense(%w1, %x_2)
20   %p_1: D2 = @dense(%w2, %ar_1)
21   %ar_2: D2 = Send(%as_2, 2)
22   %dp_1: D2 = @lossGrad(%p_1, %y_1)
23   %dw2_1: D2, %das_1: D2 = @denseGrad(%w2, %ar_1, %dp_1)
24   %dar_1: D1 = Send(%das_1, 1)
25   %dw1_1: D1, _ = @denseGrad(%w1, %x_1, %dar_1)
26   %p_2: D2 = @dense(%w2, %ar_2)
27   %dp_2: D2 = @lossGrad(%p_2, %y_2)
28   %dw2_2: D2, %das_2: D2 = @denseGrad(%w2, %ar_2, %dp_2)
29   %dar_2: D1 = Send(%das_2, 1)
30   %dw1_2: D1, _ = @denseGrad(%w1, %x_2, %dar_2)
31   // Weight update (WU)
32   %dw1: D1 = Sum(%dw1_1, %dw1_2)
33   %w1_new: D1 = Optimizer(%w1, %dw1)
34   %dw2: D2 = Sum(%dw2_1, %dw2_2)
35   %w2_new: D2 = Optimizer(%w2, %dw2)
36   return %w1_new, %w2_new
37 }
```

Figure 2. DistIR code listing for pipeline-parallel training of a 2-layer MLP model over 2 devices. The functions `@denseGrad` implements the backwards pass for an MLP layer, and `@lossGrad(%p, %y)` computes the gradient of the predictions `%p` given the labels `%y`. In DistIR, `Send` encapsulates both sending and receiving. We annotate variables with `D1` and `D2`, and color them blue and orange, to represent that they live on device 1 and 2 respectively.

operation (henceforth op), calls to other functions defined in the same module, or return statements.

DistIR, like MLIR, is designed to be extensible by being parametric on the set of primitive op types $\mathbb{O}$. The core framework requires only that ops be registered along with their function signatures. (The simulator in §3.3 requires abstract implementations and cost functions for each registered op.) DistIR's type system also allows extension with new types as required (we omit type annotations in our listings for brevity). We have instantiated DistIR with ONNX ops, corresponding backward ops, and MPI communication ops.

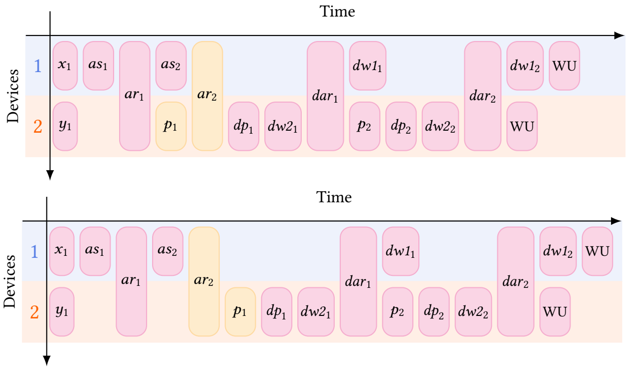

Figure 3. Traces (not-to-scale) for `@mlpPP` from Figure 2 (top) and the program obtained by swapping lines 20 (p_1) and 21 (ar_2) of `@mlpPP`. Each op is labelled by its (first) return value (with `%`s omitted) and "WU" represents the weight update.

All programs in DistIR are essentially straight-line code: there are no loops, branches, or recursive function calls. However, note that primitive ops can abstract arbitrarily complex computations, including on multiple devices, as long as we can define cost models for them.

For example, consider the program to train a 2-layer multi-layer perceptron (MLP) model over 2 devices using a pipeline parallel strategy (Figure 2). The function `@dense` represents a single layer in an MLP model, and uses primitive ops `Gemm` and `Relu` from the ONNX standard, and an `UnpackTuple` primitive to unpack a tuple of weights (for brevity). The `@mlpPP` function splits the training data into two microbatches and then executes the forward pass and backward pass on each microbatch before summing up the gradients and updating the weights. The code for each microbatch is interleaved in order to capture the efficient pipelined execution shown in the trace in Figure 3, as explained below.

2.1 Distributed Semantics

DistIR programs execute on a distributed computation model over a finite fixed set of devices $\mathbb{D}$, each of which is single-threaded and can execute at most one operation at a time. Each operation executes in a synchronous manner on a set of devices $D \subseteq \mathbb{D}$. This means that execution of the op waits until all the involved devices are free before proceeding. This set of devices can depend on the runtime input values and their locations, e.g. a `Send(%x, 2)` will run on devices 1 and 2 if its input `%x` lives on device 1.[2] The op register contains this information, along with the concrete implementations of each primitive op in $\mathbb{O}$.

[2] Since DistIR models the global computation over all devices, there is no need to have separate send and receive ops.

DistIR has an explicit *schedule*: operations execute in the program order, but consecutive operations on disjoint sets of devices execute in parallel. For example, consider `@mlpPP` from Figure 2. Assuming its input values `%w1` and `%x` (respectively, `%w2` and `%y`) live on device 1 (respectively, 2), then the first two `Split` ops execute in parallel on devices 1 and 2 (Figure 3, top). After this, the `@dense` returning `%as_1` and the `Send` returning `%ar_1` execute in sequence (because they both involve device 1), followed by simultaneous computation of `%as_2` and `%p_1` (because they involve separate devices).

However, if we swapped lines 20 (`%p_1`) and 21 (`%ar_2`), then because the `Send` involves both devices, it blocks `%p_1` on device 2 from executing until it completes (Figure 3, bottom). We see that DistIR enforces the schedule given by program order, regardless of the fact that line 20 and line 21 have no data dependencies and can be swapped without changing the program's return value.

DistIR's representation of pipeline training (`@mlpPP`) captures the distributed computation on all devices in the same function. It captures the way the inputs are split into microbatches in the first few lines; the way the model is partitioned into multiple stages using the `@dense` function; and the pipeline schedule that determines the order in which microbatches execute on a device in the program order of the multiple calls to `@dense`.

Note that we do not expect users to write DistIR code manually. Users can continue writing forward-only code (e.g., `@dense`) in a frontend like PyTorch and export it to ONNX or XLA to generate the backwards pass (e.g, `@denseGrad`), after which we import to DistIR. DistIR then distributes the code by applying transforms (resulting in, e.g, `@mlpPP`). The verbose nature of DistIR makes it easy to perform distribution, and to analyze and simulate the resulting programs.

2.2 Expressivity

DistIR is expressive enough to represent many distributed DNN training strategies of interest, including:

- data parallelism, horizontal parallelism [31], model parallelism, and multiple pipeline-parallel schedules;
- hybrid strategies such as the one-weird-trick [18];
- training optimizations such as ZeRO [29], and tensor rematerialization [12, 15];
- sparsely activated models such as GShard [21] and Switch Transformers [10].

Our evaluation of these models is ongoing.

DistIR's explicit design means that some computations are harder to model. For example, the assumption that each primitive op in DistIR is blocking synchronous means one must use lower-level communication primitives such as `Send` to model the behavior of fine-grained collective communication algorithms where some devices perform useful work before others are ready. Another common optimization is to overlap communication with computation on devices like GPUs with multiple streams. Expressing this in DistIR needs a more verbose approach of using a DistIR device per stream, and specifying that devices representing streams within the same GPU have low or zero communication cost (see §3.3).

Algorithm 1: An Abstract Interpreter for DistIR

```
given   : an abstract domain $(A, \mathbb{S})$
inputs  : a function $f$ and a list of input values $\vec{v}$
outputs : the final abstract state $\rho$
$\rho \leftarrow$ new abstract state mapping $\vec{x}$ to $\vec{v}$
foreach $op \in \vec{op}$ do
    case $op$ looks like $\vec{y} = O(\vec{x})$ do
        $\vec{w} \leftarrow$ run abstract semantics $\mathbb{S}[O](\rho(\vec{x}))$
        $\rho \leftarrow$ update $\vec{y}$ to $\vec{w}$
    case $op$ looks like $\vec{y} =$ call $foo(\vec{x})$ do
        $\rho' \leftarrow$ call Abstract Interpreter on $foo$ and $\rho(\vec{x})$
        $\rho \leftarrow$ update with $\vec{y}$ from $\rho'$
    case $op$ looks like return $\vec{x}$ do
        return $\rho, \tau$
```

3 Analyses

This section presents an analysis framework, based on abstract interpretation [7, 8], that can be used, e.g., to simulate the runtime and memory consumption of DistIR programs.

At a high-level, abstract interpretation can be thought of interpreting a DistIR program line-by-line, but with a state that maps each variable to an abstract value (such as the type `Int`) instead of a concrete value (such as 42). These abstract values represent the set of possible values that the variable can have over all executions of the program.

Abstract interpreters are parametric on the *abstract domain*, which consists of a set A of abstract values, and abstract implementations of primitive ops over this domain, represented as an abstract semantics $\mathbb{S} \colon \mathbb{O} \to A^* \to A^*$ mapping each op type to a function over abstract values. For abstract interpretation to be sound, the abstract semantics must abstract the concrete semantics (more details in [7, 8]).

Algorithm 1 gives the algorithm for abstract interpretation of a DistIR function f on input (abstract) values $\vec{v}$. It returns an abstract state $\rho \colon \mathbb{X} \to A$.

An example instantiation of abstract interpretation is type propagation. The abstract domain consists of types such as `Int`, `Float`, and `Tensor[Float]`, and the abstract implementation of each op checks that the op's inputs match the expected types and returns the type(s) of the output(s).

3.1 Lowering

We can use the abstract interpreter to perform the lowering from a DistIR program representing an entire distributed computation to the per-rank program executed by each participating device. Our abstract domain is the set of devices. The abstract implementation of each op checks that the input values live on the expected devices and then returns abstract values corresponding to the devices on which each output resides. For instance, the implementation of `MatMul` checks that all inputs are on a single device d and returns device d as the output, whereas an `Allreduce` checks that inputs are on distinct devices and returns the same list of devices. After interpretation, we project the input program to device d by filtering out all ops without inputs or outputs on device d.

3.2 Reference Executor

Another instantiation of our framework is a reference sequential executor, which can be used to check the output of an DistIR program without executing it on a cluster. This helps debug and develop parallel transforms that manipulate DistIR programs. We use an abstract domain consisting of concrete values (technically, each value represents a singleton set) and abstract implementations of each op perform a sequential version of its computation. For example, an `MPIGather` op concatenates its inputs on the specified axis.

3.3 Simulator

We can also instantiate the abstract interpreter to perform a simulation of an DistIR program in order to estimate its runtime and memory consumption.

We use a mixed abstract domain containing both concrete boolean, integer, float, and tensor values, as well as types such as `Int32`, `Float16`, and tensor types annotated with shapes, element type, and device (`Tensor[Float32, (128, 64), 0]`). The concrete values are needed to infer the output shapes of ops such as `Reshape`, and the shapes of tensors let us accurately estimate the cost of tensor ops.

The abstract implementations of ops is as follows: for most ops, such as `MatMul`, we simply propagate types and devices. Some ops, such as `Shape`, convert an abstract input like `Tensor[Float32, (128, 64), 0]` to the concrete output `[128, 64]`. An op such as `Reshape` requires a concrete shape as input and returns a tensor type with the appropriate shape. We use the input shapes to estimate the runtime of each op using analytic cost functions; but we could alternatively use profiled data. We also estimate the live memory profile for each device by calculating the memory requirement of each tensor from its shape and assuming that it is live from the time it is created until its last usage.

4 Evaluation

Our evaluation of DistIR is ongoing. We have implemented a prototype in Python consisting of the IR, the abstract interpretation framework along with instantiations to a reference executor and simulator, and a parallel transform for applying distribution in a hybrid data/horizontal/pipeline-parallel (D/H/P) space. We have yet to implement a backend for DistIR that can execute the lowered per-rank programs on real hardware (we discuss this and other ongoing improvements in §5). Thus, we investigate the following questions:

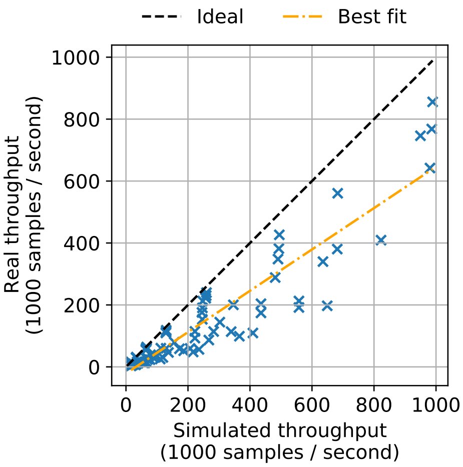

Figure 4. Comparison of DistIR's simulated throughput and actual throughput for various MLP models. Real throughput measured on similar models manually written in PyTorch.

1. Are the DistIR simulation results accurate compared to manually constructed models run on real hardware? (§4.1)
2. For large scale parallelism, do the trends reported by the DistIR simulator match intuition? (§4.2)
3. Are hybrid strategies useful for optimal distribution? (§4.3)
4. How can DistIR's simulator enable automatic optimization in the hybrid search space? (§4.3)

For all experiments we instantiate the simulator with analytic cost functions that estimate costs based on compute and network performance behavior of NVIDIA GPUs - Figure 4 models a single server with up to 4 Titan V [2] GPUs, while Figures 5 and 6 model a DGX [1] machine with 16 GPUs.

4.1 Simulator vs. Real Performance

Figure 4 shows that for deployments of up to 4 GPUs, the DistIR simulator's throughput predictions for MLP models strongly correlate with the true performance measured on real hardware using equivalent PyTorch models. We manually construct the PyTorch models and run them with 32-bit precision using the DistributedDataParallel (DDP) API. We vary the world size, batch size, model depth, and weight size for a total of 81 data points. Each physical GPU data point is the median of 100 measurements recorded after 10 warmup iterations. We find that the Pearson coefficient between the simulated predictions and true measurements is 0.9289, and the Spearman coefficient is 0.9311. This suggests that the DistIR simulator will correctly compare distributed strategies during automatic distribution.

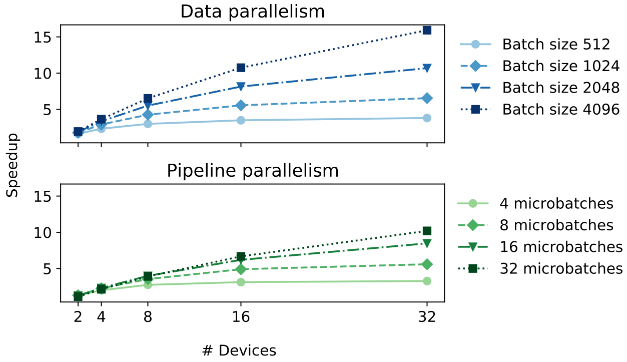

Figure 5. Simulations showing how data parallelism and pipeline parallelism scale as the parallelism degree increases for an MLP model with over 1 billion parameters. For data parallelism we compare different global batch sizes, while for pipeline parallelism we fix the global batch size and only vary the number of microbatches per minibatch.

Our simulator overestimates throughput for some data points, but this is due to differences in absolute runtime rather than differences in relative speedups as we increase the parallelism degree. This is due to a combination of: (a) missing kernel launch overheads in our per-op cost functions, (b) different communication strategies used by PyTorch's data parallelism and our implementation, (c) discrepancies in our cost model for backward ops. We plan to address these issues by adding launch overheads to our cost model, updating our data parallel transform to match PyTorch's DDP, and using profile-based costs for per-op cost models.

4.2 Large-Scale Parallelism Trends in Simulation

Figure 5 demonstrates that the DistIR simulator matches the expected behavior when applying data parallelism and pipeline parallelism in isolation to MLP models. Across all parallelism strategies, we expect better performance when the computation-to-communication ratio increases. With data parallelism we indeed observe larger speedups in throughput (samples per second processed) relative to a single-device execution as we increase the global batch size. Similarly, we see with pipeline parallelism that injecting more microbatches into the pipeline reduces "bubble" sizes and improves overall throughput.

4.3 Hybrid Parallelism and Automated Grid Search

Figure 6 illustrates the need to search through hybrid parallelism strategies in order to maximize throughput for a given model and number of devices. The plot shows speedups for training a 64-layer MLP with uniform hidden size 8192 over different numbers of devices. The lines correspond to standard strategies data, horizontal, and pipeline parallelism, and we show that the best throughput is achieved by a hybrid data-horizontal strategy. On the other hand, pipeline parallelism incurs delays due to pipeline bubbles, confirming our intuition that pipelines are not useful for models whose parameters fit on a single device.

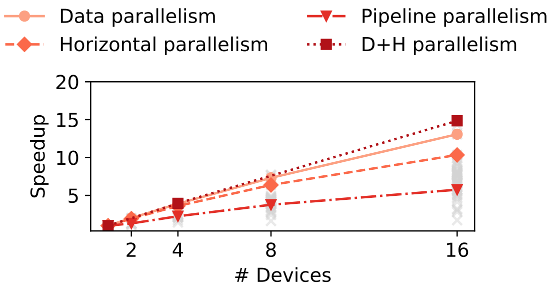

Figure 6. Simulated results from applying combinations of data, horizontal, and pipeline parallelism in varying degrees to a 4-billion parameter MLP model across several world sizes. The red points indicate the best performing configuration for each parallelism configuration and world size, while the gray points denote all other measured combinations.

We demonstrate the potential for automated distribution by performing a grid search over the D/H/P space for the MLP model above. The search space has 4 dimensions: the degree of data, horizontal, and pipeline parallelism, and, for the latter, the number of microbatches. We were able to search through the 52 resulting strategies on a 16 device setup in under a minute using a laptop with a 4-core processor. The simulator was able process DistIR programs with over 16,000 ops in under a second. This indicates it is possible to use DistIR for automatic distribution based on search algorithms. We plan to investigate more complex search algorithms as well as evaluate the results of search on actual hardware in future work (§5).

5 Discussion and Future Work

There are many promising directions for future work.

First, we plan to write an exporter that converts the per-rank DistIR programs emitted by our lowering to run on a backend such as ORT or XLA. This will allow us to compare the simulation results with profiling results on actual hardware, and creates an end-to-end DistIR workflow. We can then also use profile data from actual executions to improve our per-op cost functions.

Many recent works propose algorithms for automatic distribution, such as MCMC search [17], integer and dynamic programming [24, 33], reinforcement learning [23, 34], and custom algorithms [16, 25]. We plan to investigate these search strategies within our simulator-based search infrastructure. We also aim to extend our evaluation to other DNN models and search over optimizations such as ZeRO [29].

Many frameworks use a single-program-multiple-data (SPMD) representation for distributed computation, because it provides concise representations of common data-parallel programs [21]. In DistIR, we can outline such repetitive blocks of code into functions to reduce IR size. We are also investigating adding a `map` primitive to DistIR to make it easier to express common map-reduce patterns.

While DistIR enables composable transforms through its uniform representation, designing these transforms in practice requires careful consideration. One challenge is that each transformation must preserve a valid communication layout between different devices. This is especially difficult when constructing nested distributed strategies, as each additional transformation introduces its own collective communication which must respect prior parallelism dimensions. Another challenge is navigating the trade-off between conciseness and flexibility in the representation. For example, while the `map` primitive outlined above would avoid repeating code, it might preclude subsequent non-uniform transformations across different branches of the mapped function. We plan to explore these trade-offs in future work.

We plan to implement DistIR's analysis framework and simulator on top of MLIR. This will not only bring performance improvements, but also allow us to reuse MLIR's infrastructure, use existing generic transformations and optimization passes, as well as make it easier to instantiate DistIR with primitives from different domains.

6 Related Work

Many existing libraries for distributed DNNs [14, 24, 31] are implemented in eager frameworks such as PyTorch [26] that do not have an IR capturing the distributed computation. This makes it hard to write analyses such as a general-purpose simulator. Implementations of distribution strategies, e.g. the pipeline schedule in PipeDream [24], are typically entwined with the per-rank program code. This means a simulator would have to either be tied to a particular pipeline schedule or accept the schedule as an auxiliary input.

Graph-based frameworks such as XLA [20] and ONNX Runtime [22] lower DNN models from frontends such as Tensorflow [3] or PyTorch [26] into IRs that are capable of expressing distributed computation [10, 14, 21, 35]. Our contribution relative to these frameworks is to make explicit the semantics and scope of an IR for distribution and to build a simulation and analysis framework. We can integrate our framework with Tensorflow by importing XLA graphs into DistIR, just as we now import ONNX Runtime graphs.

There is existing work on combining distribution strategies, as well as APIs that allow users to pick from a space of strategies [21, 30]. However, as these strategies are not designed as IR transforms it is difficult to extend these to distribution strategies outside of the supported space. We plan to build a Python API for DistIR that will further lower the barrier for people developing new strategies.

DaCe [4], Lift [32], and Elevate [13] all propose IRs for representing parallel computation. However, these IRs are primarily designed for maximizing single-node parallelism rather than optimizing distributed performance for large-scale DNNs. DistIR is not a functional language, nor a classic data-flow model; rather, it is an imperative language and an SSA-based IR (though functional and data-flow models can be lowered to it). Halide [28] separates what is computed from how it is computed; we plan to investigate integrating Halide's approach in DistIR in order to make transforms more modular. TVM [6] is an end-to-end optimizing compiler for DNNs, but to our knowledge it does not consider distribution.

DayDream [36] and DNNMem [11] propose profiler-based simulators to accurately predict DNN execution time and memory respectively. However, DayDream has a special representation for optimizations and DNNMem operates over front-end model specifications, while we capture both model and distribution in a generic IR. FlexFlow [17] uses a simulator to search over a fixed strategy space but does not consider pipeline parallelism. Similarly, PipeDream-2BW [25] includes a profiler to predict performance for various pipeline parallel configurations but does not consider horizontal parallelism. DistIR's simulator is more general as it is not tied to a particular class of models or strategies.

7 Conclusion

DistIR is an efficient IR for explicit representation of distributed DNN computation. DistIR permits efficient static analyses such as simulation that accelerate manual distribution as well as enables automatic distribution via search algorithms. Expressing distribution as transformations over DistIR functions allows one to develop hybrid strategies via composition of existing strategies. Our preliminary grid search over a space of hybrid strategies demonstrates how DistIR can be used to facilitate automatic distribution.

Acknowledgments. Part of the work was done while Keshav was an intern at Microsoft, hosted by Yuan Yu. We thank our shepherd, Sam Ainsworth, as well as Trevor Gale, Fiodar Kazhamiaka, Nuno Lopes, Alberto Magni, Deepak Narayanan, Simon Peyton Jones, Deepti Raghavan, Dennis Shasha, James Thomas, Juliana Vicente Franco, Thomas Wies, and Matei Zaharia for their invaluable feedback. This research was supported in part by affiliate members and other supporters of the Stanford DAWN project—Ant Financial, Facebook, Google, Infosys, NEC, and VMware—as well as Toyota Research Institute, Northrop Grumman, Cisco, SAP, and the NSF under CAREER grant CNS-1651570. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. Toyota Research Institute ("TRI") provided funds to assist the authors with their research but this article solely reflects the opinions and conclusions of its authors and not TRI or any other Toyota entity.

References

[1] NVIDIA DGX Datasheet. URL: https://images.nvidia.com/aem-dam/Solutions/Data-Center/nvidia-dgx-a100-datasheet.pdf.

[2] NVIDIA Titan V. URL: https://www.nvidia.com/es-la/titan/titan-v/.

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[4] Tal Ben-Nun, Johannes de Fine Licht, Alexandros N Ziogas, Timo Schneider, and Torsten Hoefler. Stateful Dataflow Multigraphs: A Data-Centric Model for Performance Portability on Heterogeneous Architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2019.

[5] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language Models are Few-Shot Learners. *arXiv preprint arXiv:2005.14165*, 2020.

[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[7] Patrick Cousot and Radhia Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[8] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282, 1979.

[9] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large Scale Distributed Deep Networks. In *Advances in Neural Information Processing Systems*, pages 1223–1231, 2012.

[10] William Fedus, Barret Zoph, and Noam Shazeer. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *arXiv preprint arXiv:2101.03961*, 2021.

[11] Yanjie Gao, Yu Liu, Hongyu Zhang, Zhengxian Li, Yonghao Zhu, Haoxiang Lin, and Mao Yang. Estimating GPU Memory Consumption of Deep Learning Models. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Egineering*, pages 1342–1352, 2020.

[12] Andreas Griewank and Andrea Walther. Algorithm 799: Revolve: An Implementation of Checkpointing for the Reverse or Adjoint Mode of Computational Differentiation. *ACM Transactions on Mathematical Software (TOMS)*, 26(1):19–45, 2000.

[13] Bastian Hagedorn, Johannes Lenfers, Thomas Koehler, Sergei Gorlatch, and Michel Steuwer. A Language for Describing Optimization Strategies. *arXiv preprint arXiv:2002.02268*, 2020.

[14] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Advances in Neural Information Processing Systems*, pages 103–112, 2019.

[15] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 497–511, 2020. URL: https://proceedings.mlsys.org/paper/2020/file/084b6fbb10729ed4da8c3d3f5a3ae7c9-Paper.pdf.

[16] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 47–62, 2019.

[17] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond Data and Model Parallelism for Deep Neural Networks. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 1–13, 2019. URL: https://proceedings.mlsys.org/paper/2019/file/c74d97b01eae257e44aa9d5bade97baf-Paper.pdf.

[18] Alex Krizhevsky. One Weird Trick for Parallelizing Convolutional Neural Networks. *arXiv preprint arXiv:1404.5997*, 2014.

[19] Chris Lattner, Jacques Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: A Compiler Infrastructure for the End of Moore's Law. *arXiv preprint arXiv:2002.11054*, 2020.

[20] Chris Leary and Todd Wang. XLA: TensorFlow, compiled. *TensorFlow Dev Summit*, 2017.

[21] 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. In *International Conference on Learning Representations*, 2021. URL: https://openreview.net/forum?id=qrwe7XHTmYb.

[22] Microsoft. ONNX Runtime. URL: https://microsoft.github.io/onnxruntime/.

[23] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In *International Conference on Machine Learning*, pages 2430–2439. PMLR, 2017.

[24] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[25] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-Efficient Pipeline-Parallel DNN Training. *arXiv preprint arXiv:2006.09503*, 2020.

[26] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, pages 8026–8037, 2019.

[27] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research*, 21:1–67, 2020.

[28] Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Halide: Decoupling Algorithms from Schedules for High-performance Image Processing. *Communications of the ACM*, 61(1):106–115, 2017.

[29] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory Optimization towards Training a Trillion Parameter Models. *arXiv preprint arXiv:1910.02054*, 2019.

[30] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 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*, pages 3505–3506, 2020.

[31] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using GPU Model Parallelism.

arXiv preprint arXiv:1909.08053, 2019.

[32] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. Lift: a Functional Data-Parallel IR for High-Performance GPU Code Generation. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 74–85. IEEE, 2017.

[33] Jakub M Tarnawski, Amar Phanishayee, Nikhil Devanur, Divya Mahajan, and Fanny Nina Paravecino. Efficient Algorithms for Device Placement of DNN Graph Operators. *Advances in Neural Information Processing Systems*, 33, 2020.

[34] Siyu Wang, Yi Rong, Shiqing Fan, Zhen Zheng, LanSong Diao, Guoping Long, Jun Yang, Xiaoyong Liu, and Wei Lin. Auto-MAP: A DQN Framework for Exploring Distributed Execution Plans for DNN Workloads. *arXiv preprint arXiv:2007.04069*, 2020.

[35] Yuan Yu, Martín Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, Tim Harley, Peter Hawkins, et al. Dynamic Control Flow in Large-scale Machine Learning. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018.

[36] Hongyu Zhu, Amar Phanishayee, and Gennady Pekhimenko. Daydream: Accurately estimating the efficacy of optimizations for DNN training. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 337–352. USENIX Association, July 2020. URL: https://www.usenix.org/conference/atc20/presentation/zhu-hongyu.