SimpleFSDP: Simpler Fully Sharded Data Parallel with torch.compile

Ruisi Zhang[1,*,†], **Tianyu Liu**[2,*], **Will Feng**[2], **Andrew Gu**[2], **Sanket Purandare**[3,†], **Wanchao Liang**[2], **Francisco Massa**[2]

[1]UC San Diego, [2]Meta, [3]Harvard University
[*]Equal contribution, [†]Work done at Meta

Distributed training of large models consumes enormous computation resources and requires substantial engineering efforts to compose various training techniques. This paper presents SimpleFSDP, a PyTorch-native compiler-based Fully Sharded Data Parallel (FSDP) framework, which has a simple implementation for maintenance and composability, allows full computation-communication graph tracing, and brings performance enhancement via compiler backend optimizations.

SimpleFSDP's novelty lies in its unique `torch.compile`-friendly implementation of collective communications using existing PyTorch primitives, namely parametrizations, selective activation checkpointing, and DTensor. It also features the first-of-its-kind intermediate representation (IR) nodes bucketing and reordering in the TorchInductor backend for effective computation-communication overlapping. As a result, users can employ the aforementioned optimizations to automatically or manually wrap model components for minimal communication exposure. Extensive evaluations of SimpleFSDP on Llama 3 models (including the ultra-large 405B) using TorchTitan demonstrate up to 28.54% memory reduction and 68.67% throughput improvement compared to the most widely adopted FSDP2 eager framework, when composed with other distributed training techniques.

∞ Meta

1 Introduction

Distributed training the ever-growing large models necessitates huge computation resources Rae et al. (2021); Zhang et al. (2022); Chowdhery et al. (2023); Dubey et al. (2024) and engineering efforts Shoeybi et al. (2019); Rasley et al. (2020); Liang et al. (2024), both of which pose significant challenges as the model size scales. For example, training the Llama 3.1 405B Dubey et al. (2024) model takes 30.84 million H100 GPU hours, and PaLM-540B Chowdhery et al. (2023) model takes 9.4 million TPUv4 hours. During training, various parallelisms Huang et al. (2019); Shoeybi et al. (2019); Zhao et al. (2023), memory optimizations Chen et al. (2016); Korthikanti et al. (2023), and communication optimizations Micikevicius et al. (2017); Zhao et al. (2023); Choudhury et al. (2024) are employed to improve computation throughputs and minimize communication exposure.

Fully Sharded Data Parallel (FSDP) Zhao et al. (2023), motivated by the DeepSpeed ZeRO Rajbhandari et al. (2020), is one of the most fundamental techniques for distributed large model training. It significantly saves memory by sharding model parameters, gradients, and optimizer states across multiple devices and only gathers them when needed. As such, it is widely adopted to train large generative models Le Scao et al. (2023); Dubey et al. (2024) and has been deployed in open-source libraries, like NeMo Kuchaiev et al. (2019), DeepSpeed Rasley et al. (2020), and TorchTitan Liang et al. (2024).

FSDP is primarily developed in the PyTorch *eager* (i.e., non-compile) mode, where model operators are executed immediately after definition. It preserves debuggability and enables certain mechanisms like prefetching via backward hooks PyTorch Community (2023d), which are hard to trace in the compile mode Ansel et al. (2024). However, the eager mode impairs the training performance, as the model cannot be compiled as a whole graph, thereby losing opportunities for hardware-specific computation optimizations and efficient

1

memory management.

Prior works bringing machine learning compilers into distributed training mainly go from two directions: (1) JAX-based Bradbury et al. (2018); Xu et al. (2021) that uses XLA Sabne (2020) as the compile backend and shard tensors via user annotations; (2) PyTorch-based Liang et al. (2024), which leverages `torch.compile` Ansel et al. (2024) to trace per-device compute submodules and insert inter-module communications. JAX adopts functional programming and imposes certain constraints to ensure compatibility with the XLA backend. This greatly hinders the programmability in distributed training, which stacks many emerging techniques and requires agile development.

PyTorch-based approach Liang et al. (2024), on the other hand, only compiles the model's computation modules, as the FSDP eager-mode implementations like prefetching are hard to be traced by `torch.compile`. Hence, it loses the opportunity to compile a full model graph for communication/computation co-optimization and introduces additional codebase complexity by requiring the manual insertion of inter-module communications.

This paper presents SimpleFSDP, a PyTorch-native compiler-based FSDP framework. It features (1) **Simplicity**: users do not need to alter the eager-mode distributed training codebase while experiencing the performance enhancement from full-model compilation; (2) **Composability**: SimpleFSDP can be seamlessly integrated with emerging distributed training techniques with minimal engineering effort; (3) **Performance**: training throughputs and memory gains from full-graph tracing and compiler optimizations; and (4) **Debuggability**: SimpleFSDP exhibits usability in PyTorch eager mode, where users have the flexibility to debug and agile develop the codebase.

SimpleFSDP achieves the FSDP semantics by utilizing a few existing PyTorch primitives. First, representing the sharded per-parameter as DTensors PyTorch Community (2023b), SimpleFSDP achieves the "all-gather before usage" behavior by applying collective communications (via the DTensor `redistribute` API) as tensor parametrization. Note that the backward gradient reduce-scatter is automatically achieved as parametrization and DTensor `redistribute` are differentiable. Second, given that in parametrization PyTorch Community (2023f), parameter all-gathers are treated as activation computations, SimpleFSDP achieves the additional memory optimization of "release after forward usage, all-gather again before backward usage" by wrapping the parametrization module using activation checkpointing PyTorch Community (2023a). Since parametrization, selective activation checkpointing, and DTensor APIs are all natively supported by `torch.compile`, SimpleFSDP obtains a full graph of communication and computation operations.

SimpleFSDP introduces two optimization components in `torch.compile`'s backend TorchInductor, namely bucketing and reordering, to enhance the per-parameter sharding performance. The bucketing merges the communication operations[1] in TorchInductor to reduce the frequency of issuing base communication. The reordering pre-fetches the parameters used for computation in later stages to overlap with the current stage's computation for minimized communication exposure. Building on top of the optimizations, SimpleFSDP provides two interfaces to users to wrap the model, enabling both customization and automation. The first manual-wrapping enables users to customize the communications to bucket among modules and reorders the bucketed communication operations to reduce exposure. The auto-wrapping employs a greedy algorithm to bucket the communication operations as long as they can be overlapped by the computation operations and do not exceed memory limits.

SimpleFSDP's PyTorch-native implementation enables it to be seamlessly composed with other distributed training techniques. We demonstrate its composability with Tensor Parallel and Pipeline Parallel, meta initialization, mixed precision training, and activation checkpointing with only a few lines of code. Such composability is achieved while tracing the model's full computation-communication graph and tested on scales up to the ultra-large 405 billion parameter Llama 3.1 model Dubey et al. (2024).

In summary, our contributions are as follows:

- We introduce SimpleFSDP, a PyTorch-native compiler-based FSDP framework featuring simplicity, composability, performance enhancement, and debuggability.

- We devise SimpleFSDP highlighting (1) a unique collective communication implementation of FSDP via PyTorch primitives (parametrizations, selective activation checkpointing, and DTensor API), enabling

---

[1]The operators are lowered to IR nodes in TorchInductor. We use the two terms interchangeably throughout the paper.

full-graph tracing in model training; (2) the <mark>first-of-its-kind IR nodes bucketing and reordering in TorchInductor with flexible user interfaces</mark> (manual-wrapping and auto-wrapping) to customize and automate computation-communication overlapping.

- We perform extensive evaluations of SimpleFSDP on Llama 3 models (up to the ultra-large 405B) using TorchTitan Liang et al. (2024), demonstrating its (1) **Performance**: up to 28.54% peak memory reduction and 68.67% higher throughput improvement, compared to the most widely adopted FSDP2 eager framework PyTorch Community (2023c); (2) **Scalability and Composability**: full-graph tracing when composed with other distributed training techniques while maintaining up to 6.06% throughput gains and 8.37% memory reduction, compared to the existing best-performing sub-module compilation; (3) **Debuggability**: maintaining comparable memory and throughput in the eager mode.

2   Background and Challenges

This section first introduces techniques and related work for accelerating large models' distributed training. We then present several challenges that state-of-the-art frameworks face when supporting these techniques.

2.1   Distributed Training Large Models

Training large models in a distributed manner reduces the memory requirements per device and accelerates the computation throughputs.

Fully Sharded Data Parallel (FSDP) Zhao et al. (2023) is one of the most fundamental forms of data parallelism in distributed training. It shards model parameters, gradients, and optimizer states across multiple devices. During training, it gathers the needed parameters for computation and discards them immediately to save memory. A typical FSDP training consists of

- **Model Initialization & Parameter Sharding**: The model is wrapped into FSDP units and partitioned per the number of devices for parameter sharding. Each device only holds one of the partitions.

- **Forward Pass**: Each FSDP unit gathers the parameters from other devices and performs the computation. The parameters are discarded immediately after the computation to save memory.

- **Backward Pass**: Similar to the forward pass, each FSDP unit re-gathers the parameters and computes the gradient. The gradients are averaged and sharded across devices.

Tensor Parallel Shoeybi et al. (2019); Narayanan et al. (2021) partitions and shards tensors of an individual layer across multiple devices. Each device computes the sharded part of the layer simultaneously and concatenates them together for the outputs. *Pipeline Parallel* Huang et al. (2019); Narayanan et al. (2019); Li et al. (2021) partitions a model that cannot fit in a single device's memory into multiple stages. Each device concurrently processes a stage over multiple batches of data.

Meta initialization Zhao et al. (2023) initializes the model parameters on a meta device PyTorch Community (2023e) (an abstract device that denotes a tensor and records only metadata) rather than the actual CPU/GPU device. It reduces the time and memory required for initialization. *Mixed precision training* Micikevicius et al. (2017) reduces memory usage by training the model using 16-bit floating-point numbers. In the gradient updates, parameters are cast back to 32-bit floating-point numbers for training stability. *Activation checkpointing* Chen et al. (2016); Korthikanti et al. (2023) reduces the memory consumption by selectively storing activations at certain layers in the forward pass and recomputing the rest during the backward pass. It significantly reduces the peak memory and allows model training on memory-constraint devices.

2.2   Related Work

Machine learning compilers Chen et al. (2018); Sabne (2020); Ansel et al. (2024) accelerates model training by optimizing the computation graph execution on different target hardware devices and by performing careful memory management. In torch.compile Ansel et al. (2024), the frontend TorchDynamo captures the FX graph from the user code by just-in-time (JIT) compiling Python bytecode; the default backend TorchInductor takes the FX graph operations as input and lowers the graph to a set of intermediate representation (IR)

nodes. The IR nodes are fused and generate corresponding OpenAI Triton code Tillet et al. (2019) to write GPU kernels for more efficient execution.

Distributed training frameworks like Megatron-LM Shoeybi et al. (2019) and DeepSpeed Rasley et al. (2020) support various parallelism strategies (data, tensor, and pipeline) and memory-saving techniques (activation checkpointing and mixed precision training) to train large transformer language models at scale. They are primarily developed in the PyTorch eager mode for its debuggability and easy-to-use interface. SimpleFSDP is a PyTorch-native FSDP implementation, providing a simple plug-and-play interface, compiler-based optimizations, and composability with other parallelisms. Therefore, SimpleFSDP is orthogonal to them and can be potentially integrated into those frameworks as enhancements.

Existing auto-parallelism works Zheng et al. (2022); Lin et al. (2024); Chen et al. (2024) attempted to cover Data Parallel, but without optimizing computation-communication overlapping, thus potentially yielding sub-optimal solutions. We hope SimpleFSDP's systematic exploration of automating communication optimizations, as well as the infrastructure innovation in compiler backend, would benefit future auto-parallelism works.

2.3 Challenges

Applying machine-learning compilers to distributed training exhibits a few challenges, as outlined below.

Complexity Distributed training combines various parallelisms and memory-saving techniques to fit larger models and increase throughputs, making the codebase inherently complicated, especially when the techniques strives to improve eager-mode performance. This makes the integration and tracing of machine learning compilers difficult.

Debuggability While machine learning compilers offer performance enhancements, the debuggability from the eager mode remains crucial. It allows users to experiment with different building blocks for agile development. However, these debugging practices may violate the compilation rules, making the code untraceable Bradbury et al. (2018); Sabne (2020). As a result, a framework that preserves both debuggability under eager mode and performance enhancement from compile mode is essential.

Composability Existing parallelism implementations (e.g. DDP and FSDP in PyTorch) use backward hooks to perform efficient collective communications, making it difficult for `torch.compile` to trace. Although attempts have been made to enable full-graph compilation in those scenarios, integrating them with emerging distributed training techniques are still challenging.

3 SimpleFSDP Design

Identifying the challenges, we introduce SimpleFSDP, which maintains distributed training's **simplicity** and **debuggability**. Then, we incorporate several compiler-only optimization components to enhance SimpleFSDP's **performance**. We demonstrate SimpleFSDP's **composability** with other distributed training techniques in Section 4.

3.1 Overview

In this section, we introduce how SimpleFSDP realizes the FSDP semantics using existing PyTorch primitives, namely *parametrization* PyTorch Community (2023f) and *selective activation checkpointing* PyTorch Community (2023a), together with the DTensor abstractions PyTorch Community (2023b). All these techniques are now natively supported by PyTorch `torch.compile`.

In FSDP (or ZeRO-3), optimizer states, gradients, and model parameters are all sharded. The parameters are all-gathered in the forward pass and used for both forward and backward computation, whereas the computed gradients are reduce-scattered after backward computation for optimizer updates. SimpleFSDP shards the parameters as DTensors during model initialization and utilize PyTorch primitive parametrization and DTensor API `redistribute` to implement the all-gather in the forward pass, as in Figure 1's `ReplicateComputation`. Since DTensor's `redistribute` and parametrization are differentiable, the gradient reduce-scatter in the backward pass is automatically handled.

As a memory optimization, in the forward pass, the all-gathered parameters used for computation can be immediately released afterwards to save memory; during backward, another all-gather is issued to regather the released parameters. Such semantics can be perfectly described using activation checkpointing, which releases activations after forward computation and recomputes them before being used in the backward pass. Hence, as in Figure 1's `ReplicateComputation.replicate_compute`, the parametrization takes the sharded parameters as input and replicates the parameters as activations, where selective activation checkpointing is employed to localize the checkpointing behavior of the FSDP-related communication operators.

The implementation benefits distributed training from two aspects: (i) **Simplicity**: users only need to wrap their model with `simple_fsdp(model)`, and call `torch.compile` on the wrapped model. It allows the machine learning compiler to generate a full graph with both computation and communication operations for downstream optimizations; (2) **Debuggability**: the implementation does not alter the eager mode code execution. Users can still experiment and debug the model for agile development.

```python
import torch
from torch.distributed.tensor import distribute_tensor, Partial, Replicate, Shard
from torch.utils.checkpoint import (
    checkpoint, CheckpointPolicy, create_selective_checkpoint_contexts,
)
from torch.nn.utils.parametrize import register_parametrization

# (1) Selective Activation Checkpointing Policy
def fsdp_policy():
    def _fsdp_recomp_policy():
        def _custom_policy(ctx, func, *args, **kwargs):
            to_recompute = func in {
                torch.ops._c10d_functional.all_gather_into_tensor.default,
                torch.ops._c10d_functional.wait_tensor.default,
            }
            return (
                CheckpointPolicy.MUST_RECOMPUTE
                if to_recompute
                else CheckpointPolicy.MUST_SAVE
            )
        return _custom_policy
    return create_selective_checkpoint_contexts(_fsdp_recomp_policy())

# (2) Collective Communication Implementation
class ReplicateComputation(torch.nn.Module):
    def replicate_compute(self, x):
        return x.redistribute(
            placements=(Replicate(),),
        ).to_local(grad_placements=(Partial(reduce_op="avg"),))

    def forward(self, x):
        return checkpoint(
            self.replicate_compute, x, use_reentrant=False, context_fn=fsdp_policy
        )

# (3) Parameter Sharding and Parametrization Registration
def simple_fsdp(model):
    for mod in list(model.modules()):
        params_dict = dict(mod.named_parameters(recurse=False))
        for p_name, p in params_dict.items():
            if p is not None and p.numel() > 0:
                mod.register_parameter(
                    p_name,
                    torch.nn.Parameter(distribute_tensor(p, placements=(Shard(0),)))
                )
                register_parametrization(
                    mod, p_name, ReplicateComputation(), unsafe=True,
                )
    return model
```

Figure 1 SimpleFSDP's frontend implementation.

3.2 Optimizations

The graph traced from SimpleFSDP is lowered to a set of IR nodes in TorchInductor. This alone does not yield optimized training performance, as the communication and computation operations from `ReplicateComputation` are shared per-parameter in sequential order, and all of the communication operations are exposed. As depicted in Figure 2, we introduce two optimizations in TorchInductor to enhance the distributed training performance: (1) **Bucketing** to group and merge communication IR nodes to reduce the frequency of issuing base communication; (2) **Reodering** to prefetch the communication IR nodes for overlapping with current computation.
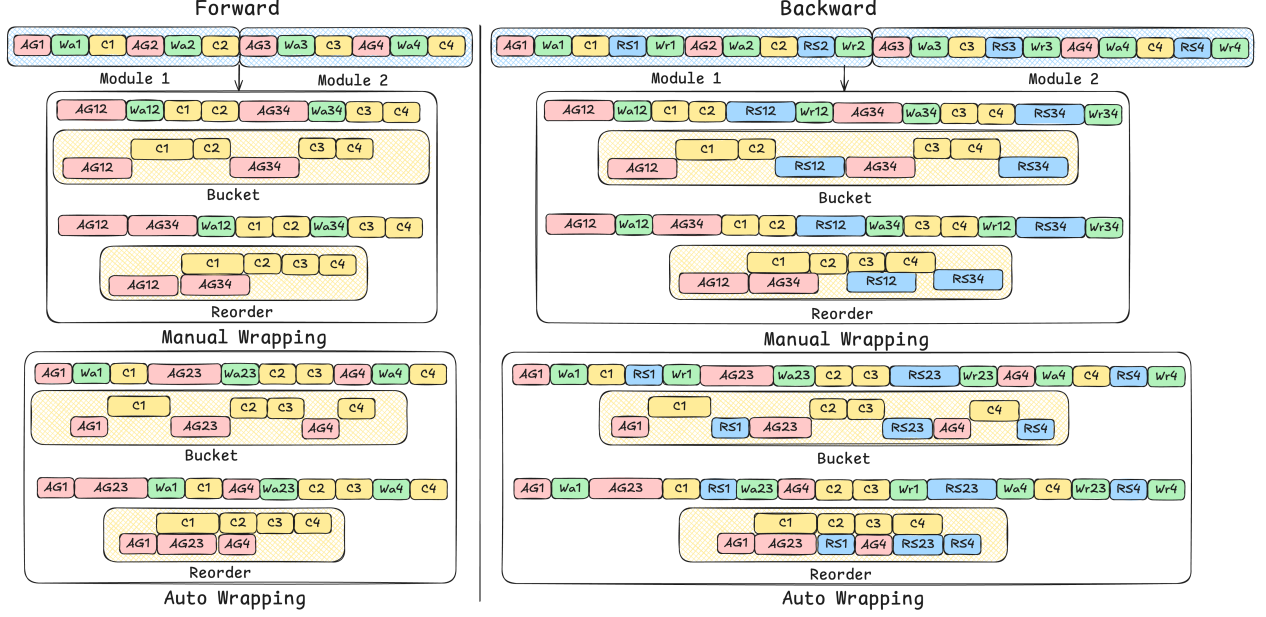
Figure 2 An overview of SimpleFSDP's optimizations and model wrapping. The left side is the forward pass, and the right side is the backward pass. We show the IR node scheduling in TorchInductor and the corresponding execution order in GPU in the yellow box. The blue box indicates the IR nodes are from the same module. In the **Manual Wrapping**, the all-gathers (AG) and reduce-scatters (RS) from the same module are bucketed as new communication IR nodes. Then, the bucketed communication and computation are reordered to enable communication prefetch during the current computation. In the **Auto Wrapping**, the all-gather and reduce-scatter are bucketed as long as the bucketed communication can be overlapped by the current computation and does not exceed the memory limit. Then, the bucketed communication and computation are reordered to hide communication exposure.

3.2.1 Bucketing

The communication cost between two devices comprises a base latency for establishing the communication and a transmit latency proportional to the transmitted word size NVIDIA (2024). By bucketing the communication IR nodes, SimpleFSDP issues the base communication once for all of the bucketed nodes and thereby reduces the overall communication time.

As in Figure 2, the individual all-gather/reduce-scatter IR node reads the data and issues the communication. To bucket the all-gather IR nodes $AG1$ and $AG2$, SimpleFSDP allocate a bigger buffer that flattens and concatenates the tensor from each individual all-gather. The new all-gather $AG12$ and all-gather-wait $Wa12$ are created to gather the bigger buffers from other devices and copy out the gathered data based on their original tensor size.

To bucket reduce-scatter, SimpleFSDP splits the obtained gradient into chunks based on world size and concatenates the gradients from the individual reduce-scatter $RS1$ and $RS2$'s data to create a bigger buffer. A new reduce-scatter $RS12$ and reduce-scatter-wait $Wr12$ are created to average the buffer data gathered from other devices. The gradients are read out from $RS12$ to update the local model weight.

3.2.2 Reordering

The collective communication all-gather and reduce-scatter are asynchronous, allowing it to occur concurrently with the computation on different CUDA streams. In Figure 2, in the forward pass, each computation has an all-gather and an all-gather-wait to gather the data; in the backward pass, each computation has additional reduce-scatter and reduce-scatter-wait to update the gradient. Reordering the IR nodes ensures the communications can overlap by the computations and thereby reduces the communication exposure.

As in Figure 2, we use the manual wrapping as an example; the reordering process is as follows: (1) In the forward pass, the $AG34$ is reordered in front of $Wa12$. It allows $AG34$ to overlap with compute $C1$; (2) In the backward pass, the $AG34$ is placed after $Wa12$, enabling $AG34$ to overlap with $C1$. The $Wr12$ is placed

before $RS34$, such that $RS12$ can overlap with the later compute $C3$ and $C4$.

There are additional computations to copy out data from all-gather and reduce-scatter after the respective wait IR node. In the forward pass, by placing $AG34$ before $Wa12$, $AG34$ can further overlap with the compute to copy out data from the bigger buffer after $Wa12$. In the backward pass, $RS12$ is before $Wa34$, making it overlap with the compute to copy out data from $AG34$; thereby, AG can be placed after Wa as the copy-out compute has already been overlapped.

3.3 Model Wrapping

Building on top of the optimizations in Section 3.2, SimpleFSDP provides two wrapping interfaces, namely, manual-wrapping and auto-wrapping, to bucket communication IR nodes together and reorder them for overlapping with computation operations.

The manual-wrapping buckets communication IR nodes based on pre-defined module lists. It provides the same functionality as those in FSDP2 PyTorch Community (2023c), where users can customize module wrapping after model definition. The auto-wrapping provides a more fine-grained and automatic bucketing interface, where no input from users is required. As the model is shared per parameter, SimpleFSDP employs a greedy algorithm to atomically bucket communication IR nodes from each parameter for minimized exposure.

3.3.1 Manual-wrapping

In TorchInductor, each IR node contains metadata that traces its original module name. It enables SimpleFSDP to construct a mapping between module names and their corresponding IR nodes. Thus, users can customize the wrapping rules by providing a list of module names. SimpleFSDP then wraps the communication/computation nodes between these modules. As in Figure 2, the communication IR nodes from module 1 and module 2 are bucketed separately and reordered to overlap the bucketed communication.

3.3.2 Auto-wrapping

Profiling The profiling algorithm estimates the IR nodes' communication and computation time in TorchInductor. For the computation node, SimpleFSDP converts the FakeTensor (containing tensor metadata without actual data) into real PyTorch Tensors. It executes the computation node's Python kernel with these real Tensors as input and records the CUDA event time T_c and the peak memory M_c. For the communication node, we formulate the estimated communication time as $T_m = \alpha + \beta n$ where n denotes the transmitted word size and α, β are the transmit parameters NVIDIA (2024).

Wrapping The wrapping algorithm automatically buckets the communication IR nodes to minimize communication exposure while keeping the memory within the limit.

Variable	Definition
T_m^{AG}	current step's bucketed AG's communication time
T_c	current step's computation time
M_c	next step's peak computation memory
T_m^{RS}	last step's bucketed RS's communication time
T_{mi}^{AG}	i-th AG's communication time
T_{ci}	time to compute the parameters pre-fetched by i-th AG
M_{ci}	peak memory to compute the parameters pre-fetched by i-th AG
T_{mi}^{RS}	time to reduce-scatter the gradient for parameters in i-th AG

Table 1 Variable definition

We show an example of the bucketing decision process in Algorithm 1, where the variables are defined in Table 1. In the forward pass, we decide if the i-th all-gather node can be bucketed with the previous all-gather as long as it satisfies (1) Time Constraint: the communication time after bucketing the i-th all-gather, denoted as $T_{(m+mi)}^{AG}$, can be overlapped by the current step computation (line 4) and (2) Memory Constraint: the

7

pre-fetched computation memory in the next step does not exceed the memory limit M_{max} (line 5). Otherwise, the i-th all-gather will not be bucketed with the previous all-gather.

In the backward pass, the i-th all-gather nodes are bucketed with the previous all-gather as long as it satisfies (1) Time Constraint: the previous step's reduce-scatter T_m^{RS} and the current step's all-gather when bucketing the i-th all-gather, denoted as $T_{(m+mi)}^{AG}$, can be overlapped by the current step's computation (line 10), and (2) Memory Constraint: the pre-fetched computation memory in the next step does not exceed the memory constraint M_{max} (line 11). The corresponding reduce-scatter IR nodes of the all-gathers are bucketed as well. Otherwise, the i-th all-gather will not be bucketed with the previous all-gather.

Algorithm 1: Auto Wrapping Algorithm

1: **Input:** T_m^{AG}, T_m^{RS}, T_c, M_c, T_{mi}^{AG}, T_{mi}^{RS}, T_{ci}, M_{ci}
2: **Output:** True for bucket; False for not bucket
3: **if** $isForward$ **then**
4:     $timeConstraint = (T_{(m+mi)}^{AG} \leq T_c)$
5:     $memConstraint = (M_c + M_{c(i)} \leq M_{max})$
6:     **if** $timeConstraint$ **and** $memConstraint$ **then**
7:         **return** True
8:     **end if**
9: **else if** $isBackward$ **then**
10:     $timeConstraint = (T_m^{RS} + T_{(m+mi)}^{AG} \leq T_c)$
11:     $memConstraint = (M_c + M_{c(i)} \leq M_{max})$
12:     **if** $timeConstraint$ **and** $memConstraint$ **then**
13:         **return** True
14:     **end if**
15: **end if**
16: **return** False

3.4 User interface

SimpleFSDP provides simple plug-in-play interface. After defining the parallelism configs, users employ the `simple_fsdp` API to wrap the model with SimpleFSDP, as introduced in Section 3.1. Then, the `torch.compile` API compiles the model, tracing both communication and computation operations.

The model wrapping, including reordering and bucketing, is handled by the TorchInductor backend. No additional modifications are required for the existing distributed training codebase. It greatly reduced the burden of maintaining distributed training code while providing performance gains.

`fullgraph=True` generates a full model graph. If the model has untraceable content, e.g., data-dependent control flow, setting `fullgraph=False` splits the graph into several subgraphs for SimpleFSDP to optimize.

```
1 torch._inductor.config.simplefsdp.bucket_mode = "auto"
2 torch._inductor.config.simplefsdp.enable_reorder = True
3 model = simple_fsdp(model)
4 model = torch.compile(model, fullgraph=True)
```

4 Composability

SimpleFSDP is natively implemented with *DTensor*, *parametrization* and *selective activation checkpointing*, making it easy to be integrated with techniques in Section 2.1 to train large models with few lines of code.

Meta initialization During model weight initialization on the meta device, SimpleFSDP disables the all-gather parametrization to reduce the time and memory required to load the model.

Mixed precision training The `param_dtype` and `reduce_dtype` are parsed into the DTensor redistribution. During training, the model parameters are cast to `param_dtype`, while the gradients are cast to `reduce_dtype`. Mixed

precision training is enabled by setting `param_dtype` to the 16-bit floating point and `reduce_dtype` to the 32-bit floating point.

```python
1  def replicate_compute(self, x):
2      output = x.redistribute(
3          placements=(Replicate(),),
4          forward_dtype=self.param_dtype,
5          backward_dtype=self.reduce_dtype,
6      ).to_local(grad_placements=(Partial(reduce_op="avg"),))
```

Tensor Parallel In parametrization computation, a model parameter can be initialized as a 2D DTensor, doubly sharded on both Data Parallel (DP) and Tensor Parallel (TP) dimensions. During computation, it is first redistributed (via an all-gather) on the DP sub-mesh, and then represented as a sharded DTensor on the TP sub-mesh, ready for the following TP computations.

Pipeline Parallel The model is partitioned into several submodules, with each device receiving a copy of the assigned submodule. SimpleFSDP wraps the received submodule before computation. No additional code is required to ensure compatibility with Pipeline Parallel.

Activation checkpointing Similarly, the activation checkpointing is applied before SimpleFSDP. After defining which operations to recompute, SimpleFSDP wraps the model and applies the additional checkpointing policies to the FSDP communication operations. No additional code is needed to compose with activation checkpointing.

5 Experiments

Infrastructure We build SimpleFSDP in PyTorch TorchInductor with $\sim$ 2K LoC. The benchmarking is performed on TorchTitan Liang et al. (2024). Our evaluation environment includes a CPU/GPU cluster with 16 nodes. Each node has 8 NVIDIA H100 GPUs, and the intra-node is connected via NVLink Wei et al. (2023).

Target Models and Metrics We evaluate SimpleFSDP on Llama 3.1 series Dubey et al. (2024) models of various sizes. The details are in Table. 2.

| Model	Layers	Model Dim.	FFN Dim.	Head Num.
8B	32	4,096	14,336	32
70B	80	8,192	28,672	64
405B	126	16,384	53,248	128

Table 2 Llama 3.1 Model Configurations.

The performance is evaluated by training the models on the C4 dataset Zhu et al. (2024). The reported metrics are (i) **Token-per-second (TPS)**, the training throughput as the number of processed tokens per second; (2) **Memory**, the peak training memory.

Baselines We compare SimpleFSDP with PyTorch FSDP2 PyTorch Community (2023c) developed by the official PyTorch team. It is an improved implementation of FSDP Zhao et al. (2023) by offering lower memory consumption and higher throughput.

- FSDP2-eager: The target model's submodules are wrapped as FSDP2 units and executed in the PyTorch eager mode. It is the widely adopted distributed training setting.

- FSDP2-compile: Each target model's submodules is compiled with TorchInductor before being wrapped as an FSDP2 unit. It offers a stronger baseline by applying `torch.compile` to computation operations without handling collective communication tracing.

For fair comparisons, in Section 5.1- 5.2, SimpleFSDP employs manual-wrapping to bucket the communication per transformer-block, and FSDP2-compile compiles the computation operations in each transformer-block region. In Section 5.3, we study the auto-wrapping performance.
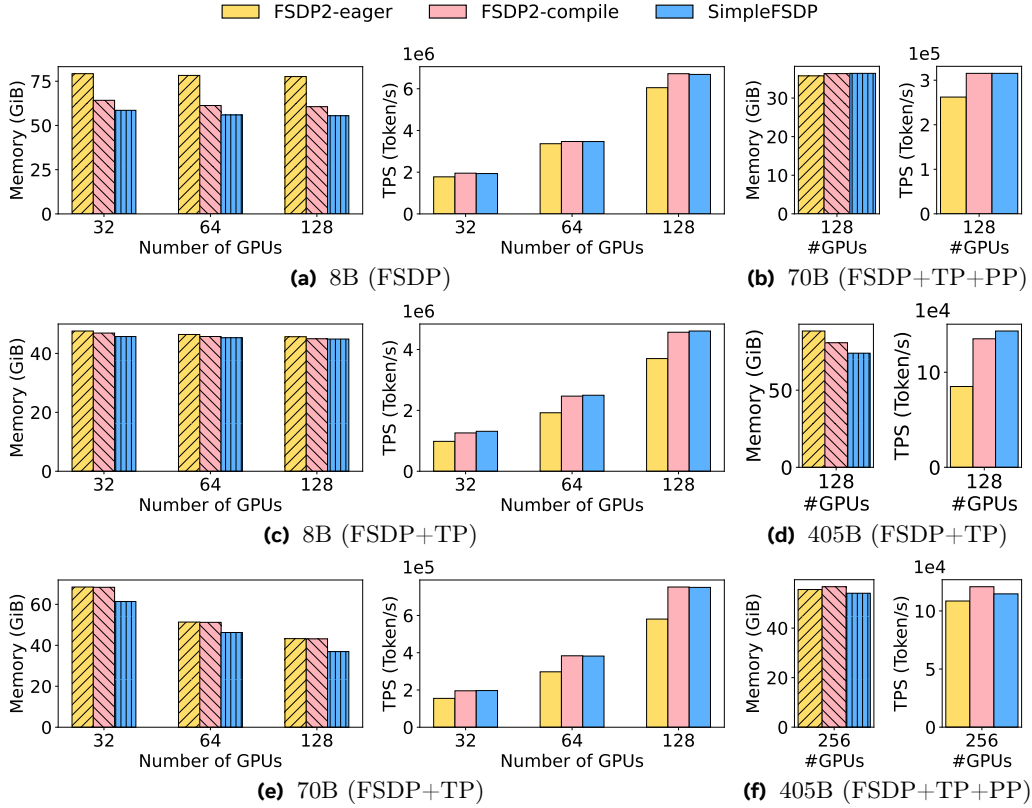
Figure 3 SimpleFSDP performance on LLaMA-3 8B, 70B, and 405B models when training on different numbers of H100 GPUs. We report the peak memory in GiB and the throughputs in TPS (tokens/s).

5.1 SimpleFSDP Performance

Figure 3a shows the memory and throughput for Llama 3.1 8B trained with FSDP on 32, 64, and 128 H100 GPUs. Compared to FSDP2-eager, SimpleFSDP on average saves 27.72% peak memory and improves the throughput by 7.49%. The performance gains are primarily from the memory optimizations, IR node fusions, etc, in `torch.compile` that make the training more efficient. Compared to FSDP-compile, SimpleFSDP still maintains 8.65% peak memory reduction by tracing the full-graph, where `torch.compile` obtains a global view and allocates the memory better. The transformer-based LLaMA models are computation-intensive, meaning the communication is fully overlapped when only FSDP is applied. As such, both SimpleFSDP and FSDP2-compile hit the throughput upper bound for training the Llama 3.1 8B model, resulting in on-par throughput performance.

Where memory savings come from We identify the following three main reasons for SimpleFSDP's memory savings. (1) SimpleFSDP works at a tensor-level, allowing a finer granular memory management (e.g. all-gathered parameters can be released sooner), compared with FSDP2's module-level behavior. This gives memory advantage to SimpleFSDP on the Llama model. (2) `torch.compile` makes different decisions on what activations to save for FSDP2 block-level compilation and SimpleFSDP whole-model compilation. The FSDP2-compile will save additional transposed tensors from scaled dot product attention (SDPA) outputs, whereas SimpleFSDP only saves SDPA outputs. (3) SimpleFSDP manages bucketing on the same CUDA stream as the rest computes. FSDP2 uses multiple streams, which has a throughput benefit but can cause memory allocation fragmentation.

5.2 SimpleFSDP Composability and Scalability

In this section, we present the performance after composing SimpleFSDP with Tensor Parallel and Pipeline Parallel. All of the models employ full activation checkpointing (AC) and mixed precision training. In the compile mode, we apply Asynchronous Tensor Parallel Wang et al. (2022).

10

2D Composability The Llama 3.1 8B and 70B models are trained with FSDP and Tensor Parallel Shoeybi et al. (2019). The Tensor Parallel degree is set to 8, and the batch size is set to 16 and 8, respectively. Figure 3c and 3e show the performance when training Llama 3.1 8B and 70B models on 32, 64, and 128 GPUs.

SimpleFSDP can be integrated with Tensor Parallel without degrading the performance. As seen, when training the 8B model, compared to eager mode, SimpleFSDP averagely saves 2.67% peak memory and improves the throughputs by 29.35%. Apart from the IR node fusion, SimpleFSDP benefits from the Asynchronous Tensor Parallel Wang et al. (2022) that overlaps the submatrix multiplication with communication operations. Compared to FSDP2-compile, by tracing the full-graph, SimpleFSDP further demonstrates 2.09% throughput improvement.

As the model size becomes larger, the full graph traced by SimpleFSDP provides more memory optimization opportunities in `torch.compile` and yields better performance. As in the 70B model, SimpleFSDP improves FSDP2-eager's throughputs by 28.26% and reduces the training peak memory by 11.61%. While both SimpleFSDP and FSDP-compile hit the throughput upper bound, SimpleFSDP further reduces 11.40% peak memory from the full-graph tracing.

3D Composability The Llama 3.1 70B models are trained with FSDP, Tensor Parallel Shoeybi et al. (2019), and Pipeline Parallel Huang et al. (2019). The Tensor Parallel degree is set to 8, the Pipeline Parallel degree is set to 8, and the batch size is set to 16.

SimpleFSDP can be integrated with Pipeline Parallel without performance degradations. As seen in Figure 3b, SimpleFSDP improves the throughput by 20.31% compared to FSDP2-eager while incurring less than 1GiB peak memory overhead. The models are partitioned into smaller submodules, with each device receiving a subgraph of the full model. SimpleFSDP optimizes a smaller graph compared to 2D settings, thus achieving comparable performance with FSDP2-compile. Notably, SimpleFSDP traces a communication-computation partitioned subgraph on each device, making it possible to overlap the bubbles from 3D training and bring opportunities for future optimizations.

Scalability We show the Llama 3.1 405B model 2D parallelism training performance in Figure 3d and 3D parallelism performance in Figure 3f. The Tensor Parallel degree is set to 8, the Pipeline Parallel degree is set to 16, and the batch size is set to 2 in 2D parallelism and 16 in 3D parallelism.

SimpleFSDP is scalable and maintains the performance enhancement when training ultra-large models. As seen, compared to FSDP2-eager, SimpleFSDP improves the throughput by 68.67% and 5.66% on 2D and 3D parallel, respectively. Besides, SimpleFSDP reduces the memory by 16.26% and 2.64% on 2D and 3D parallelism. Compared to FSDP2-compile, by tracing the full model graph, SimpleFSDP improves the throughput by 6.06% on 2D parallel and reduces the memory by 8.37% and 4.63% on 2D and 3D parallel, respectively.

The throughput gains and memory savings become more significant when training large models at scale: (1) Large model training requires millions of GPU hours Dubey et al. (2024); Chowdhery et al. (2023), thereby even small per-iteration throughput gains substantially reduce overall training time.; (2) The memory savings allow for larger batch sizes training per iteration, which in turn increases the throughput.
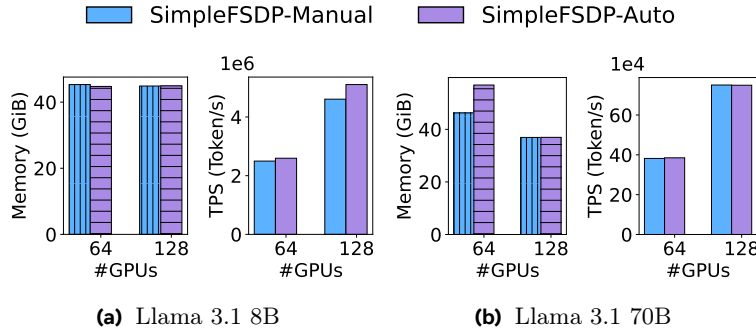
5.3 Auto-Wrapping Performance

The auto-wrapping performance is in Figure 4. The communication operations are fully-overlapped when training Llama models with only FSDP. Hence, We show the performance when training Llama 3.1 8B and 70B on 2D parallelism, where Asynchronous TP communication is exposed. Other settings follow Section 5.2.

SimpleFSDP-Auto reduces the exposed communication during large model training and does not require manual wrapping plans defined by the users. As seen, in 8B model, SimpleFSDP-Auto achieves $\sim 7.34\%$ throughput improvement over SimpleFSDP-Manual while maintaining comparable memory consumption. It means more communications are overlapped by SimpleFSDP-Auto, providing both automation and performance enhancement to users.

However, we also provide one case where SimpleFSDP-Auto provides 0.8% throughput improvement when training Llama 3.1 70B models on 64GPUs but incurs 10.61GiB memory overhead. It is primarily because

11

SimpleFSDP-Auto prioritizes minimizing the exposed communication, and the memory threshold we set is larger than the peak memory in SimpleFSDP-Manual. As a result, SimpleFSDP-Auto gives a suboptimal solution and scarifies the memory for throughput improvement. The major focus of SimpleFSDP is providing an elegant way of tracing a full graph with both communication and computation operations for downstream applications. We leave exploring algorithms to generate more optimal overlapping plans as future work.

Figure 4 Auto-Wrapping performance when training Llama 3.1 8B and 70B models on different numbers of H100 GPUs.

5.4 Analysis and Ablation Study

This subsection presents analysis of how different optimization components impact SimpleFSDP's performance. By default, we train the Llama 3.1 8B model on 8 H100 with only FSDP and set the batch size to 1. Additional ablation studies are in the appendix.

Debuggability Apart from the compile mode performance gains, SimpleFSDP exhibits usability in the PyTorch eager mode. It offers users the flexibility to print variables and experiment with various building blocks for debugging and agile development. As is shown in Table 3, SimpleFSDP achieves comparable memory consumption and throughput to FSDP2, which is primarily developed for the eager mode[2]. Notably, SimpleFSDP offers eager-mode debuggability with greater simplicity, composability, and performance gains in compile mode.

| Method	AC	TPS ↑ (Token/s)	Memory↓ (GiB)
FSDP2-eager	None	47,088	86.75
SimpleFSDP	None	46,936	91.91
FSDP2-eager	Full	37,504	37.35
SimpleFSDP	Full	38,504	29.80

Table 3 SimpleFSDP's debuggability in the eager mode. AC is for activation checkpointing.

Training convergence SimpleFSDP and its optimization components will not alter the training convergence. Figure 5 compares the loss plots of FSDP2 and SimpleFSDP when training the Llama 3.1 8B model for 1,000 epochs on 8 H100 GPUs. As seen, the similar loss convergence for both methods demonstrates that SimpleFSDP maintains model convergence and training stability.

Compilation time The time takes to compile the Llama 3.1 8B model is in Table 4. We split the total compilation time to reorder and bucket IR nodes in SimpleFSDP, and the rest time to compile the model in TorchInductor. As seen, compiling SimpleFSDP incurs negligible overhead compared to the overall training time, making it efficient.

The effectiveness of reorder and bucket Table 5 shows the impact of reordering and bucketing on single and multi-node training. In single-node, reordering enables the computation and communication to happen concurrently. It increases the training throughput and memory usage. Bucketing further increases memory by

---
[2]SimpleFSDP with no AC incurs slightly higher memory than FSDP2 due to the frontend design choice to make the codebase simpler and is not fundamental; after all, we would rely on the compiler for good performance.
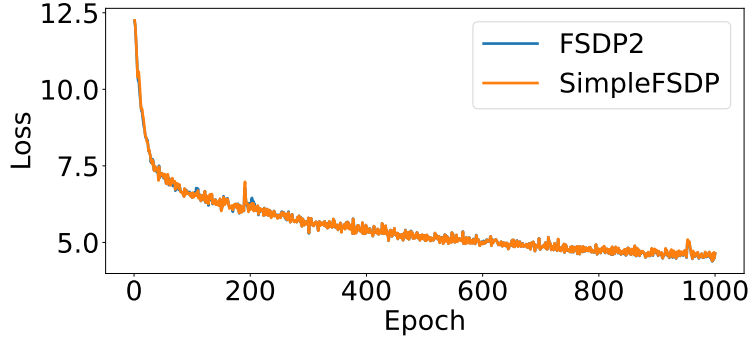
Figure 5 Loss curve of FSDP2 and SimpleFSDP on Llama 3.1 8B.

	Bucket	Reorder	Others	Total
SimpleFSDP-Manual	0.71	0.13	23.64	24.48
SimpleFSDP-Auto	3.87	0.16	21.53	25.56

Table 4 Compilation time (in second) on Llama 3.1 8B.

grouping the IR nodes for computation and communication. However, it slightly reduces throughput due to the additional time needed for copy-in/copy-out data from the buffer, which is more significant compared to the intra-node base latency that bucketing aims to optimize.

In the multi-node setting, reordering similarly increases the throughput and memory compared with the vanilla setting. Bucketing further increases the throughput by merging the IR nodes to reduce the frequency needed to establish inter-node base communication, which is non-negligible compared to intra-node base latency.

	1 node		8 nodes	
	TPS ↑	Memory↓	TPS ↑	Memory↓
	(Token/s)	(GiB)	(Token/s)	(GiB)
vanilla	50,976	**67.26**	333,440	**56.42**
+ reorder	**54,544**	68.72	404,032	57.88
+ bucket	49,168	69.06	405,632	58.15
+ reorder & bucket	52,480	69.08	**428,352**	65.74

Table 5 Effectiveness of reorder and bucket.

The effectiveness of reordering all-gather before/after the last all-gather-wait We analyze the impact of reordering all-gather before or after the last all-gather-wait in Table 6. In the forward pass, placing all-gather before the last all-gather-wait results in higher throughputs, as the compute to copy-out data from the last all-gather is overlapped by the reordered all-gather.

In the backward pass, placing reduce-scatter-wait before the next reduce-scatter already optimizes the throughput gains that could have been achieved by the all-gather. Therefore, placing the all-gather after the last all-gather-wait will save the memory slightly.

6 Discussion

In this section, we discuss SimpleFSDP in different use cases and potential future work.

Graph breaks in model tracing While SimpleFSDP obtains a full graph with both communication and computation operations, it does not require users to write code adhering to strict compilation constraints (e.g., avoiding data-dependent control flow or variable printing). Built upon `torch.compile`, when encountering non-traceable

Forward		Backward		TPS ↑	Memory↓
before	after	before	after	(Token/s)	(GiB)
✓		✓		51,912	69.08
✓			✓	**52,480**	**69.08**
	✓	✓		51,680	68.09
	✓		✓	51,776	68.09

Table 6 The effectiveness of reordering all-gather before/after the last all-gather-wait.

operations, the full graph is split into several subgraphs, each with communication and computation operations. SimpleFSDP then optimizes each subgraph individually.

Limitation While SimpleFSDP demonstrates promising performance, there are some cases where the auto-wrapping yields slightly worse performance than manual-wrapping. We found the discrepancy is due to inaccurate communication time estimation. Currently, the profiling algorithm only models the transmitted word size, whereas other factors like network topology Nedić et al. (2018) are not taken into consideration. Besides, the greedy algorithm does not consider the overall nodes' runtime, which might result in suboptimal solutions. We leave these as future work to further improve SimpleFSDP's auto-wrapping algorithm. We note that such accurate runtime estimations are not SimpleFSDP-specific but would benefit any estimation-based algorithmic decision-making, e.g., auto-parallelism.

Future work Tracing a full graph with computation and communication operations brings the potential for many downstream works. For instance, researchers can reduce the exposed communication bubbles He et al. (2021); Zhang et al. (2023); Feng et al. (2024) in distributed training or heterogeneous environments by auto-wrapping the computation/communication operations.

SimpleFSDP optimizes training performance by bucketing and reordering the IR nodes to minimize the communication exposure in multi-dimensional parallelisms. Given that our work sets up the necessary infrastructure to enable such communication optimizations in PyTorch compiler, we hope to promote more research in the related area. For example, with the type of computation-communication overlapping for Data Parallel in SimpleFSDP, auto-parallelism engines Zheng et al. (2022); Chen et al. (2024); Lin et al. (2024) can now aim for better plan execution and thus improved decision-making.

7 Conclusion

We present SimpleFSDP, a PyTorch-native compiler-based FSDP framework. It features simplicity for distributed training codebase maintenance, composability with other efficient training techniques, performance enhancement from full graph tracing, as well as debuggability and programmability from the PyTorch eager mode. Building on top of the unique parametrizations implementation of all-gather to checkpoint parameters, SimpleFSDP buckets and reorders the IR nodes for minimized communication exposure and provides customized and automated model wrapping interfaces to users. Extensive evaluations demonstrate SimpleFSDP's efficacy in throughput gains, memory saving, and scalability toward tracing ultra-large models.

Acknowledgements

.

References

Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. PyTorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 929–947, 2024.

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. http://github.com/jax-ml/jax.

Hongzheng Chen, Cody Hao Yu, Shuai Zheng, Zhen Zhang, Zhiru Zhang, and Yida Wang. Slapo: A schedule language for progressive optimization of large deep learning model training. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 1095–1111, 2024.

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

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.

Arnab Choudhury, Yang Wang, Tuomas Pelkonen, Kutta Srinivasan, Abha Jain, Shenghao Lin, Delia David, Siavash Soleimanifard, Michael Chen, Abhishek Yadav, et al. MAST: Global scheduling of ML training across geo-distributed datacenters at hyperscale. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 563–580, 2024.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. PaLM: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Weiqi Feng, Yangrui Chen, Shaoyu Wang, Yanghua Peng, Haibin Lin, and Minlan Yu. Optimus: Accelerating large-scale multi-modal LLM training by bubble exploitation. *arXiv preprint arXiv:2408.03505*, 2024.

Chaoyang He, Shen Li, Mahdi Soltanolkotabi, and Salman Avestimehr. Pipetransformer: Automated elastic pipelining for distributed training of large-scale models. In *International Conference on Machine Learning*, pages 4150–4159. PMLR, 2021.

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. *Advances in neural information processing systems*, 32, 2019.

Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems*, 5:341–353, 2023.

Oleksii Kuchaiev, Jason Li, Huyen Nguyen, Oleksii Hrinchuk, Ryan Leary, Boris Ginsburg, Samuel Kriman, Stanislav Beliaev, Vitaly Lavrukhin, Jack Cook, et al. NeMo: a toolkit for building ai applications using neural modules. *arXiv preprint arXiv:1909.09577*, 2019.

Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. Bloom: A 176b-parameter open-access multilingual language model. 2023.

Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. TeraPipe: Token-level pipeline parallelism for training large-scale language models. In *International Conference on Machine Learning*, pages 6543–6552. PMLR, 2021.

Wanchao Liang, Tianyu Liu, Less Wright, Will Constable, Andrew Gu, Chien-Chin Huang, Iris Zhang, Wei Feng, Howard Huang, Junjie Wang, et al. TorchTitan: One-stop PyTorch native solution for production ready LLM pre-training. *arXiv preprint arXiv:2410.06511*, 2024.

Zhiqi Lin, Youshan Miao, Quanlu Zhang, Fan Yang, Yi Zhu, Cheng Li, Saeed Maleki, Xu Cao, Ning Shang, Yilei Yang, et al. nnScaler: Constraint-guided parallelization plan generation for deep learning training. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 347–363, 2024.

Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

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.

Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on GPU clusters using Megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

Angelia Nedić, Alex Olshevsky, and Michael G Rabbat. Network topology and communication-computation tradeoffs in decentralized optimization. *Proceedings of the IEEE*, 106(5):953–976, 2018.

NVIDIA. NVIDIA collective communication library (NCCL), 2024. https://github.com/NVIDIA/nccl.

PyTorch Community. PyTorch Checkpoint, 2023a. https://pytorch.org/docs/stable/checkpoint.html.

PyTorch Community. PyTorch DTensor RFC, 2023b. https://github.com/pytorch/pytorch/issues/88838. GitHub Issue.

PyTorch Community. PyTorch FSDP2 RFC, 2023c. https://github.com/pytorch/pytorch/issues/114299. GitHub Issue.

PyTorch Community. PyTorch backward hook tutorial, 2023d. https://pytorch.org/docs/stable/generated/torch.Tensor.register_hook.html. PyTorch Tutorial.

PyTorch Community. PyTorch meta device, 2023e. https://pytorch.org/docs/stable/meta.html. PyTorch Tutorial.

PyTorch Community. PyTorch Parametrization, 2023f. https://pytorch.org/tutorials/intermediate/parametrizations.html.

Jack W Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, et al. Scaling language models: Methods, analysis & insights from training gopher. *arXiv preprint arXiv:2112.11446*, 2021.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

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.

Amit Sabne. XLA: Compiling machine learning for peak performance. *Google Res*, 2020.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

Philippe Tillet, Hsiang-Tsung Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 10–19, 2019.

Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechtman, Dehao Chen, Karthik Srinivasa Murthy, Marcello Maggioni, Qiao Zhang, et al. Overlap communication with dependent computation via decomposition in large deep learning models. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 93–106, 2022.

Ying Wei, Yi Chieh Huang, Haiming Tang, Nithya Sankaran, Ish Chadha, Dai Dai, Olakanmi Oluwole, Vishnu Balan, and Edward Lee. 9.3 NVLink-C2C: A coherent off package chip-to-chip interconnect with 40gbps/pin single-ended signaling. In *2023 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 160–162. IEEE, 2023.

Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, et al. GSPMD: general and scalable parallelization for ml computation graphs. *arXiv preprint arXiv:2105.04663*, 2021.

Ruisi Zhang, Mojan Javaheripi, Zahra Ghodsi, Amit Bleiweiss, and Farinaz Koushanfar. AdaGL: Adaptive learning for agile distributed training of gigantic GNNs. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2023.

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. OPT: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. Pytorch FSDP: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277*, 2023.

Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating inter-and intra-operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, 2022.

Wanrong Zhu, Jack Hessel, Anas Awadalla, Samir Yitzhak Gadre, Jesse Dodge, Alex Fang, Youngjae Yu, Ludwig Schmidt, William Yang Wang, and Yejin Choi. Multimodal C4: An open, billion-scale corpus of images interleaved with text. *Advances in Neural Information Processing Systems*, 36, 2024.