Slapo: A Schedule Language for Progressive Optimization of Large Deep Learning Model Training

Hongzheng Chen*
Cornell University
Ithaca, New York, USA
hzchen@cs.cornell.edu

Cody Hao Yu†
Boson AI, Inc
Santa Clara, California, USA
cody@boson.ai

Shuai Zheng†
Boson AI, Inc
Santa Clara, California, USA
shuai@boson.ai

Zhen Zhang
Amazon Web Services
Santa Clara, California, USA
zhzhn@amazon.com

Zhiru Zhang
Cornell University
Ithaca, New York, USA
zhiruz@cornell.edu

Yida Wang
Amazon Web Services
Santa Clara, California, USA
wangyida@amazon.com

Abstract

Recent years have seen an increase in the development of large deep learning (DL) models, which makes training efficiency crucial. Common practice is struggling with the trade-off between usability and performance. On one hand, DL frameworks such as PyTorch use dynamic graphs to facilitate model developers at a price of sub-optimal model training performance. On the other hand, practitioners propose various approaches to improving the training efficiency by sacrificing some of the flexibility, ranging from making the graph static for more thorough optimization (e.g., XLA) to customizing optimization towards large-scale distributed training (e.g., DeepSpeed and Megatron-LM).

In this paper, we aim to address the tension between usability and training efficiency through separation of concerns. Inspired by DL compilers that decouple the platform-specific optimizations of a tensor-level operator from its arithmetic definition, this paper proposes a schedule language, Slapo, to decouple model execution from definition. Specifically, Slapo works on a PyTorch model and uses a set of schedule primitives to convert the model for common model training optimizations such as high-performance kernels, effective 3D parallelism, and efficient activation checkpointing. Compared to existing optimization solutions, Slapo *progressively* optimizes the model "as-needed" through high-level primitives, and thus preserving programmability and debuggability for users to a large extent. Our evaluation results show that by scheduling the existing hand-crafted optimizations in a systematic way using Slapo, we are able to improve training throughput by up to 2.92× on a single machine with 8 NVIDIA V100 GPUs, and by up to 1.41× on multiple machines with up to 64 GPUs, when compared to the out-of-the-box performance of DeepSpeed and Megatron-LM.

*Part of the work done while interning at Amazon.
†Work done while at Amazon.

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0385-0/24/04
https://doi.org/10.1145/3620665.3640399

CCS Concepts: • **Software and its engineering → Domain specific languages**; • **Computing methodologies → Neural networks**.

Keywords: Schedule Language, Distributed Training, Compiler Optimization, Deep Learning, Large Language Models

ACM Reference Format:
Hongzheng Chen, Cody Hao Yu, Shuai Zheng, Zhen Zhang, Zhiru Zhang, and Yida Wang. 2024. Slapo: A Schedule Language for Progressive Optimization of Large Deep Learning Model Training. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3620665.3640399

1 Introduction

The demand of large deep learning (DL) models is surging in recent years as they demonstrate dominating model accuracy on a range of tasks in natural language processing (NLP) [3, 5, 10, 12] and computer vision [13, 34, 66]. These models are normally invented in user-friendly DL frameworks like PyTorch [42] with dynamic model graphs[1], which by design lacks sufficient optimization for high-performance execution. This issue becomes more and more critical as the size of models grows exponentially and so does the time of training.

In order to reduce the model training time, developers propose various kinds of optimization. The first type of optimization is implemented manually in different layers of model training, such as inserting high-performance kernels [11, 29, 41, 55] for computationally intensive operators

[1]Dynamic graph DL frameworks construct the model graph on the fly when executing its forward computation instead of constructing the graph ahead of time.

on specific devices (e.g., NVIDIA GPUs), employing data, tensor, and pipeline parallelism [38, 50, 55], as well as activation checkpointing [8, 22, 26], to efficiently distribute the training across multiple devices. However, manual optimization introduces the following two challenges.

Challenge 1: Generality – Incorporating the above optimizations requires making intrusive changes to the model implementation, which means that the optimization is not easy to generalize to other models. A new model, even with minimal change from the old one, may not be able to directly reuse the old optimization. In addition, the optimized model becomes platform-specific, requiring developers to maintain multiple implementations to serve all requirements (e.g., training on different platforms and deploying on non-GPU devices).

Challenge 2: Ease of Tuning – In practice, an optimization scheme has a number of configurations to tune (e.g., pipeline stages, number of activation checkpoints) to get a combination that results in the best performance. Developers need to identify tunable configurations in the implementation and modify the model to expose them for effective tuning. This process can be tedious and error-prone especially when the model definition is closely tied to optimizations.

In addition to manual optimization, the other set of optimization approaches converts the DL model into a number of *static graphs* and leverages DL compilers to automatically apply optimizations. For example, JAX [4] is a DL framework powered by a compiler XLA [18]. JAX traces the entire model to obtain a whole graph statically, on top of which the compiler can perform aggressive optimizations such as operator fusion, expression simplification, and even 3D parallelism [71]. Similarly, the recent release PyTorch 2.0 [43] provides a compiler interface to trace PyTorch dynamic graph executions and construct static graphs in `torch.fx` [52] for optimizations. While automatic optimization requires minimal engineering effort from model developers and addresses some of the challenges mentioned above, it also introduces two new challenges.

Challenge 3: Programmability – Working on static model graphs is limited by the requirement that everything must be statically analyzable and deterministic. Frameworks may impose constraints on the users to facilitate the conversion to static graphs. For example, JAX programming model requires pure Python functions, no in-place updates, etc., so developers may need to rewrite the model to meet these constraints in order to make it runnable [4]. For another example, PyTorch 2.0 cannot trace through the collective operators like `all_reduce` which are essential for distributed training [43]. Moreover, it is usually non-trivial for developers to control or configure the optimizations in fine granularity, such as disabling certain rules, or excluding certain operators from a compiler pass.

Challenge 4: Debuggability – To make model implementation easy to understand and maintain, model developers usually implement layer modules (e.g., convolutional, fully connected, and attention layers) as building blocks, and use them to compose a model *hierarchically*. However, to expand the scope of optimization and improve performance, DL compilers operating on a static model graph often flatten the hierarchy to create a single-level dataflow graph, and rewrite certain operators (e.g., decomposing the `batch_norm` op into a number of smaller ones). This prevents developers from understanding and troubleshooting performance or convergence issues, as the optimized model may bear little resemblance to the original model implementation.

To address the challenges mentioned above, we propose *Slapo*[2], a **S**chedule **LA**nguage for **P**rogressive **O**ptimization, designed for DL frameworks with dynamic model graphs. Slapo has the following major features.

Decouple model execution from its definition. To address Challenge 1, we decouple model execution (named "schedule") from its definition. As a result, model developers can maintain the same model implementation, and performance engineers can optimize a model- or platform-specific schedule in a separate place. This idea is inspired by well-known domain-specific compilers – Halide [49] and Apache TVM [7] – which propose widely adopted schedule languages that decouple tensor operator scheduling from its arithmetic definition.

Auto-tuning. A separate schedule also enables massive auto-tuning opportunities. Similar to AutoTVM [9], Slapo provides a programming interface that allows developers to specify a set of tuneable knobs to form an efficient tuning space. The tuning space can then be explored by Slapo auto-tuner to realize the optimal configuration, which addresses Challenge 2. Along this direction, Slapo can also enable auto-scheduling as Ansor [70], and this is our planned future work.

Progressive optimization. Slapo incorporates a "trace by need" approach that only traces a desired module to be a static graph for compiler-based aggressive optimizations. The traced part can be expanded or shrunk progressively as needed. Developers explicitly call the scheduling primitives to realize this, addressing Challenge 3.

Structure-preserving scheduling. Model developers usually define building blocks (e.g., convolutional or attention layers), and then compose them together to form a complete model. Consequently, developers often leverage this structure to analyze and debug the model. Slapo preserves this hierarchy when constructing the schedule (see §3.1 for details), so that developers can easily locate the module and apply scheduling. Also, as the model structure is preserved and optimization can be progressively applied, it facilitates the users to debug any performance and convergence issue, and a verifier (§3.5) is provided to further aid debugging, addressing Challenge 4.

In summary, we make the following contributions:

[2]https://github.com/awslabs/slapo

- We propose Slapo, a schedule language that decouples model execution from definition, and preserves model structure hierarchy to enable progressive optimization.
- We define a comprehensive set of schedule primitives for Slapo to cover prevalent optimizations in distributed training, and provide an extensible infrastructure for users to easily incorporate their own optimizations.
- We design and implement a lightweight auto-tuner for further reducing the efforts required to identify the optimal schedule configurations for training efficiency.
- We evaluate Slapo by training popular deep learning models with billions of parameters and compare Slapo with the state-of-the-art (SOTA) distributed training frameworks such as DeepSpeed [51] and Megatron-LM [55]. With minimal programming effort, Slapo is capable of scheduling the existing hand-crafted optimizations to achieve up to 2.92× speedup on a single machine with 8 NVIDIA V100 GPUs, and up to 1.41× speedup on multiple machines with 64 V100 GPUs, when compared to the out-of-the-box best baselines.

2 Background and Motivation

In this section, we first introduce common practices of improving a DL model training efficiency for dynamic graphs (e.g., PyTorch), followed by an end-to-end motivating example to illustrate the challenges of this process.

2.1 Efficient Model Training

High-performance kernel libraries. To achieve high efficiency in deep learning model training, it is straightforward to leverage efficient kernels specifically optimized for particular hardware platforms (e.g., NVIDIA GPUs, Google TPUs, and AWS Trainium). For example, there are many libraries [11, 29, 41, 55] that provide efficient CUDA kernel implementations. These libraries encapsulate kernels as DL framework-compatible modules for developers to replace the native implementation in their models. In the case where there are no existing CUDA implementations, developers may leverage compiler-based solutions, such as TorchScript [46] and TorchInductor [45], to generate a kernel.

Activation checkpointing. Apart from compute optimization techniques, memory footprint optimization is also essential for training large models. A large portion of memory consumption in the training process is contributed by forward activation tensors that are stored for the later gradient calculation. By checkpointing some activation tensors and recomputing the rest of them in backward propagation, we are able to significantly reduce memory footprint, and thus support a larger batch size and higher training throughput. This approach is called activation checkpointing and is originally proposed by [8]. Furthermore, existing works [22, 26, 27, 65] also demonstrate that by carefully selecting which activations to checkpoint, we are capable of better utilizing device memory and achieving an even better throughput.

Parallelism in distributed training. When the model is too large to fit in a single device, training it in parallel in a distributed environment is inevitable. The parallelism techniques are usually classified into three types: data parallelism, tensor parallelism, and pipeline parallelism. Both tensor and pipeline parallelism belong to a larger class called model parallelism. *Data parallelism* partitions training data, so each device trains the replicated model with different data [1, 31, 50], and aggregates their partial gradients for parameter updating. Since data parallelism replicates an entire model on each device, it is insufficient when the model size (i.e., total memory consumption of its parameters) is too large to fit on a single GPU. In this case, *tensor parallelism*, takes another approach by partitioning the model parameter tensor onto multiple devices [55]. However, it requires developers to explicitly use collective communication operators to scatter tensors and aggregate partial results. For example, Megatron-LM [55] is a widely used PyTorch-based framework that provides manual parallelized Transformer models [61] and is adopted to train extremely large models [67]. Finally, *pipeline parallelism* [20, 38] partitions models by layers and groups them into a series of stages. By putting each stage on a different device, we can overlap the computation of multiple data batches. To ensure correctness and performance, pipeline parallelism needs a specialized runtime to schedule and synchronize data. These techniques are not mutually exclusive and can be combined. Combining all of them is known as *3D parallelism* [39].

2.2 A Motivating Example

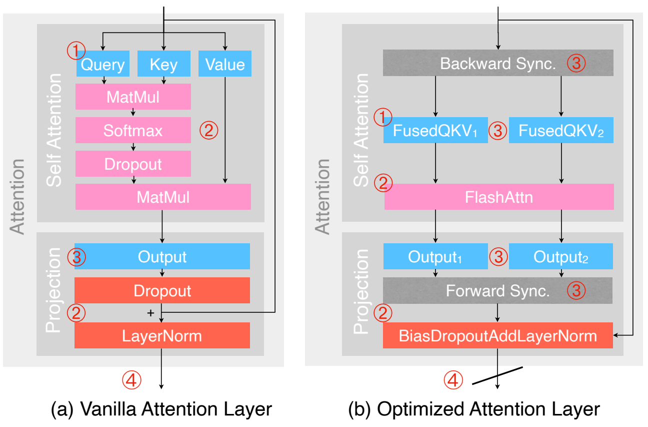


Figure 1. An attention layer in BERT. — The `Query`, `Key`, `Value`, and `Output` are `nn.Linear` modules containing learnable weights and biases. ① - ④ indicate optimization points.

In this subsection, we use BERT [12] model implementation from HuggingFace Hub [62] to showcase how the above

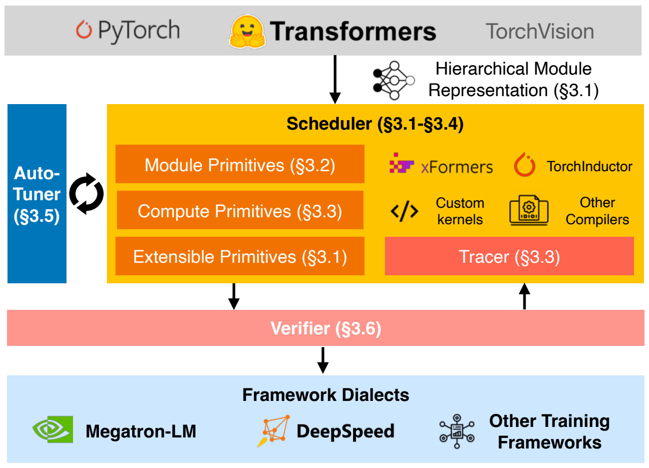


Figure 2. Overview of Slapo.

techniques are applied to a DL model for efficient training. Fig. 1(a) depicts the architecture of attention layer [61], the core and most time-consuming module in BERT. An attention layer is composed of two submodules – `SelfAttention` and `Projection`. We conduct a few steps to progressively improve the training efficiency of this attention layer and show the resulting implementation in Fig. 1(b).

① **Fuse QKV.** By default, the `Query`, `Key`, `Value` in `SelfAttention` are three standalone `nn.Linear` modules, which may incur extra kernel launch overheads. We replace them with a single `nn.Linear` module with their parameters concatenated, as shown in the following code snippet.

```
def __init__(self, ...):
-   self.query = nn.Linear(n_hidden, n_head)
-   self.key = nn.Linear(n_hidden, n_head)
-   self.value = nn.Linear(n_hidden, n_head)
+   self.qkv = nn.Linear(n_hidden, n_head * 3)
def forward(self, hidden_states, ...):
-   query = transpose(self.query(hidden_states))
-   key = transpose(self.key(hidden_states))
-   value = transpose(self.value(hidden_states))
+   qkv = transpose(self.qkv(hidden_states))
+   query, key, value = split(qkv, 1, dim=-1)
```

② **Use efficient kernels.** The pink blocks in Fig. 1 are the core attention computation, which is also the bottleneck of performance and memory footprint. A recent work flash attention [11] proposes to compute the attention in a block-by-block fashion, so only a block of the intermediate attention tensor is generated at a time, significantly reducing the peak memory consumption, and thus can improve the training efficiency by enlarging the batch size. The following code snippet shows how we replace the existing attention with the flash attention implementation provided by xFormers [29]. The transpose and reshape operations are simplified.

```
def forward(self, hidden_states, ...):
-   attn = query @ key.T
-   attn = attn / attn.shape[-1] ** 0.5
-   attn = dropout(softmax(attn), p)
-   attn = attn @ value
+   attn = xformers.ops.mem_eff_attention(...)
```

Note that this flash attention implementation only supports the latest NVIDIA GPUs with Volta, Ampere, and Hopper architectures, so once this kernel is used, the model is no longer compatible with other platforms.

Another optimization opportunity is the pattern in `Projection`. By default, the bias addition operation is contained in the `Output` module. A more aggressive and more efficient way is to use a DL compiler (e.g., TorchScript [46], TorchInductor [45]) to fuse the pattern `BiasAdd-Dropout-ResidualAdd-LayerNorm` into a single kernel, as suggested by [53].

③ **Tensor parallelism.** We then partition the `FusedQKV` and `Output` parameters onto multiple devices. Given the input of the attention module X, the weights of `FusedQKV` (A) and `Output` (B), we have self-attention function f:

$$f(XA)B = f\left(X\begin{bmatrix}A_1 & A_2\end{bmatrix}\right)\begin{bmatrix}B_1\\B_2\end{bmatrix} = f(XA_1)B_1 + f(XA_2)B_2\,.$$

Accordingly, we follow the convention of Megatron-LM [55] to shard the weights of `FusedQKV` in columns and shard the weights of `Output` in rows. We illustrate the latter in the following code snippet.

```
def __init__(self, ...):
-   self.output = nn.Linear(n_hidden, n_hidden)
+   new_size = n_hidden // world_size
+   self.output = nn.Linear(new_size, n_hidden)
def forward(self, hidden_states):
    out = self.output(hidden_states)
+   dist.allreduce(out)
```

Since the output tensor only holds partial results after sharding, we need to conduct `all_reduce` to aggregate outputs from different devices.

④ **Pipeline parallelism.** To pipeline a BERT model and execute it on a SOTA pipeline runtime, we have to further manually partition the model to a sequence of sub-models (each of them includes a series of attention layers) by rewriting the top module[3].

The above process poses a generality issue. Although developers have spent efforts on identifying and optimizing the performance bottleneck of a model with semantics preserved, this effort is hard to be reused by another model. Furthermore, the above improved model is no longer compatible with a single device, unless we add control logic to only enable model parallelism on multi-GPU environments or maintain a separate single-device version. It also creates a barrier for the model deployment after training, because a model implementation with custom efficient kernels and parallelism may not be recognized by inference compilers.

[3]The code example is omitted due to page limit.

Table 1. Comparison among Slapo and other systems. — DP, TP, and PP denote data, tensor, and pipeline parallelism, respectively. PT denotes PyTorch. Model coverage means how easily developers can leverage the programming system to optimize a new model.

	Frame-work	Model Coverage § 3.1	3D parallelism DP	TP	PP	Subgraph Opt. § 3.3.1	Act. Ckpt. § 3.3.1	Extensible Opt. § 3.1
			§ 3.2.2 & § 3.3.2					
Megatron-LM [55]	PT	○	✓	✓	✓	○	◐	✗
DeepSpeed [51]	PT	●	✓	✗	✓	○	◐	✗
Alpa [71]	JAX	●	✓	✓	✓	◐	○	✗
`pt.compile` [43]	PT	◐	✗	✗	✗	●	○	✗
Slapo	PT	●	✓	✓	✓	●	●	✓

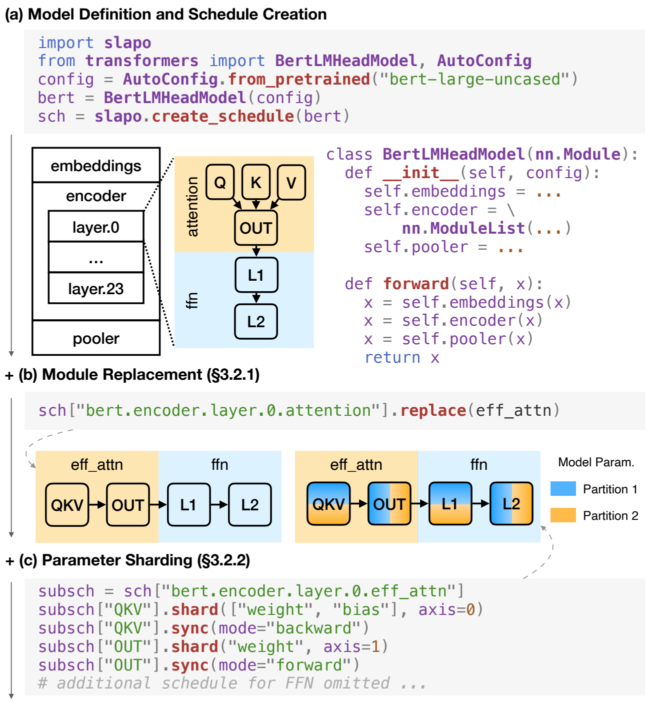
In essence, the above pain points are the result of tightly coupling model definition and training/platform-specific optimizations. This motivates us to propose a schedule language that decouples model execution (i.e., schedule) from definition and provides easy-to-use primitives for optimizing large model training. In fact, the idea of decoupling optimization has been widely accepted in DL compilers [2, 7, 49], and opens opportunities for auto-tuning [7] and auto-scheduling [70].

3 Slapo Design

This section presents the design of Slapo, our schedule language to progressively optimize DL model training using proposed primitives. Slapo decouples model definition from its training execution strategy for better portability. Slapo also abstracts out the common optimization techniques using a set of primitives to apply (or un-apply) one by one, lowering the bar for performance engineers to try out different optimization ideas. Furthermore, Slapo makes it possible to automate the performance tuning via hyperparameter search.

Fig. 2 illustrates the overview of Slapo. Slapo accepts a deep learning model implementation in a DL framework with dynamic graphs (e.g., PyTorch [42]) and parses the original model execution as its default schedule. Then, developers make use of the schedule primitives for optimizations on top of the default schedule. Slapo provides a comprehensive set of primitives that cover the prevalent distributed training optimizations, and Table 1 compares Slapo with other frameworks.

We define the schedule language in §3.1, and present the scheduling in various scenarios in §3.2 and §3.3. The scheduling strategy can be auto-tuned to search for a configuration that achieves the best performance (§3.4). Meanwhile, Slapo adopts a verifier (§3.5) to ensure the functional correctness of all schedules. After the schedule is determined and applied, the scheduled model can be trained on the runtime of the original DL framework (e.g., PyTorch), or if needed, on the dedicated runtime of existing distributed systems such as DeepSpeed [51] pipeline.



```python
import slapo
from transformers import BertLMHeadModel, AutoConfig
config = AutoConfig.from_pretrained("bert-large-uncased")
bert = BertLMHeadModel(config)
sch = slapo.create_schedule(bert)
```

```python
class BertLMHeadModel(nn.Module):
  def __init__(self, config):
    self.embeddings = ...
    self.encoder = \
        nn.ModuleList(...)
    self.pooler = ...

  def forward(self, x):
    x = self.embeddings(x)
    x = self.encoder(x)
    x = self.pooler(x)
    return x
```

```python
sch["bert.encoder.layer.0.attention"].replace(eff_attn)
```

```python
subsch = sch["bert.encoder.layer.0.eff_attn"]
subsch["QKV"].shard(["weight", "bias"], axis=0)
subsch["QKV"].sync(mode="backward")
subsch["OUT"].shard("weight", axis=1)
subsch["OUT"].sync(mode="forward")
# additional schedule for FFN omitted ...
```

Figure 3. An example of scheduling modules and parameters of a BERT model. — `ffn` is the Feed-Forward Network. `eff_attn` refers to the replaced attention module. The weight matrix has a shape of `(output_dim, input_dim)`. Thus, sharding the weight matrix at `axis=0` is equivalent to partitioning the output dimension.

3.1 Schedule Language

Motivated by §2.2, our goal is to let developers optimize models in a few lines of code without changing the model definition itself. Fig. 3 presents the Slapo schedule language with the BERT model from HuggingFace Hub [62] as an example. As shown in Fig. 3(a), most DL model developers define models with a hierarchical structure for better readability and easy maintenance. The `__init__` constructor defines the configurations, submodules, and learnable parameters, and the `forward` method defines the forward computation (the backward computation is generated by the framework with automatic differentiation [42]). Developers can then pass the created model into Slapo and create a default *schedule* that specifies *how* to execute the model in the framework. The schedule preserves the hierarchical structure, and `create_schedule` is applied recursively to all the submodules, so that developers can easily apply schedule primitives at arbitrary levels.

Built-In Primitives. As shown in Table 2, we categorize schedule primitives in two sets: whether or not they require a static graph to be generated. On one hand, when scheduling at the module level, such as enabling activation checkpointing, replacing with an efficient alternative, and sharding

Table 2. A summary of Slapo built-in schedule primitives.

Primitives with Dynamic Graphs	Primitives with Static Graphs
`.replace(new_mod)`	`.replace(new_mod, subgraph)`
`.shard(param_name, axis)`	`.fuse(subgraph, compiler)`
`.sync(type)`	`.pipeline_split()`
`.checkpoint()`	`.checkpoint(subgraph)`

learnable parameters, we do not change the computation specified in the `forward` method. As a result, the schedule primitives in the left column of Table 2 do not require a static graph, thus maximally avoiding the limitation of tracers. We present the details of this scheduling in §3.2. On the other hand, scheduling the computation, such as operator fusion and pipeline parallelism, has to manipulate the `forward` method. Thus, the schedule primitives in the right column of Table 2 require the computation to be in a static graph, so we have to use `.trace()` prior to applying these primitives, as presented in §3.3. These primitives have covered existing optimizations ranging from parallelism schemes and compiler optimizations, which are general enough to support efficient training of different models, as demonstrated in §5.
Extensible Primitives. In addition to the predefined primitives, users have the flexibility to incorporate their custom training optimization as a schedule primitive in Slapo. This can be achieved by inheriting the provided base primitive class as shown below. During program execution, Slapo dynamically registers the user-defined primitives, enabling seamless collaboration with other built-in primitives, the verifier, and the auto-tuner.

```python
@slapo.register_primitive()
class UserDefinedPrimitive(slapo.Primitive):
  def __init__(self, name):
    ... # Initialize data structure and preconditions
  def apply(self, sch, **kwargs):
    ... # Transformation on the schedule
```

3.2 Schedule Modules and Parameters

We first present scheduling a module and its parameters, which typically does not change the computation and thus does not require static graphs.

3.2.1 Schedule Modules For important workloads such as `attention` in Fig. 3(a), researchers or hardware vendors may manually implement efficient kernels [11, 25]. These highly customized, hand-crafted kernels sometimes could outperform the ones generated by DL compilers. With Slapo, we can use `.replace(new_module)` primitive to replace a native implementation with an efficient one, where `new_module` is the custom module to be replaced, as shown in Fig. 3(b).

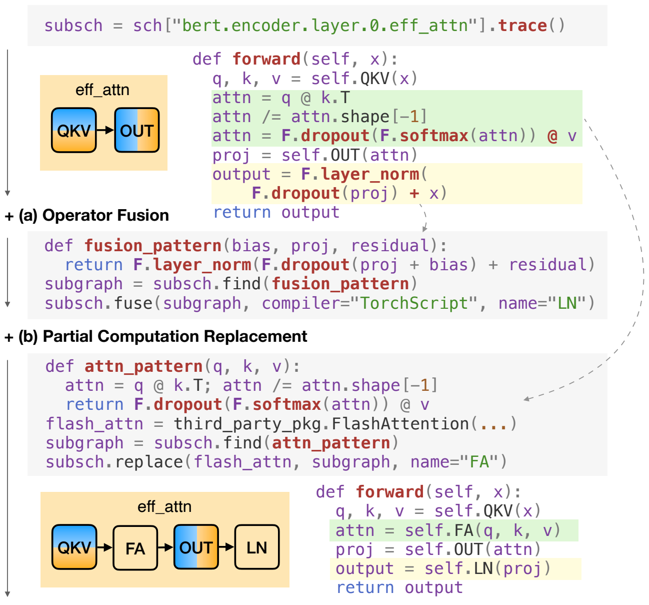
Additionally, activation checkpointing is another important feature for large model training, as mentioned in §2.1. Existing frameworks [51, 55] implement fixed strategies of checkpointing in their model definition and instantiate each layer with the same configuration, thus making it difficult to incorporate other checkpointing techniques [22, 27]. Slapo decouples the checkpointing logic and offers a `.checkpoint()` primitive that can explicitly control whether a module should be checkpointed. Consequently, Slapo enables developers to flexibly adjust the number of checkpoints via our schedule primitive or leverage the auto-tuner for better memory and throughput trade-offs.

3.2.2 Parameter Sharding In ③ of §2.2, we introduced the steps to enable tensor parallelism, which involves sharding parameters and aggregating outputs. This process is commonly known as the main challenge in adapting models for distributed execution. The manual management of partitioning and communication within the model leads to a non-executable partitioned model when the number of devices changes, as well as makes synchronization with upstream model changes difficult. While Megatron-LM [55] provides tensor parallel modules for users, they are limited to specific models. If user-defined module operators differ from predefined modules, tensor parallelism cannot be utilized for distributed training.

In contrast, Slapo overcomes these limitations by enabling users to shard a parameter using the `.shard(param, axis)` primitive and aggregate results using the `.sync(type)` primitive. The `type` can be "forward" (aggregate the forward activations) or "backward" (aggregate the gradients). Notice Slapo can efficiently capture the parameter and axes information covering the entire space of model partition, including 3D parallelism [39, 55] and other complicated parallelism schemes that an automatic compiler [71] supports. These primitives can be applied to arbitrary models and parameters, effectively addressing generality issues. It also does not require the model to be traceable since sharding does not modify the computation graph. Fig. 3(c) shows that implementing a complex tensor parallel program only requires five lines of schedule code without modifying the model definition. Slapo automatically shards parameters for different distributed environments and inserts synchronization points based on users' annotations. Meanwhile, we employ a verifier (§3.5) to check correctness after scheduling. In the future, we plan to develop an auto-scheduler that automatically generates these primitives.

3.3 Schedule Computations

The prerequisite of scheduling computations is tracing the `forward` method of the target module, and constructing a static graph in a certain intermediate representation (IR). There are several approaches to obtaining the static graph IR. First, *run-with-dummy-data* [46] is an approach that directly executes the method with dummy inputs and captures all executed operators in order. Second, *AST-analysis* [46] directly analyzes Python abstract-syntax-tree (AST) to obtain the



```
subsch = sch["bert.encoder.layer.0.eff_attn"].trace()
```

```
def forward(self, x):
  q, k, v = self.QKV(x)
  attn = q @ k.T
  attn /= attn.shape[-1]
  attn = F.dropout(F.softmax(attn)) @ v
  proj = self.OUT(attn)
  output = F.layer_norm(
      F.dropout(proj) + x)
  return output
```

+ (a) Operator Fusion

```
def fusion_pattern(bias, proj, residual):
  return F.layer_norm(F.dropout(proj + bias) + residual)
subgraph = subsch.find(fusion_pattern)
subsch.fuse(subgraph, compiler="TorchScript", name="LN")
```

+ (b) Partial Computation Replacement

```
def attn_pattern(q, k, v):
  attn = q @ k.T; attn /= attn.shape[-1]
  return F.dropout(F.softmax(attn)) @ v
flash_attn = third_party_pkg.FlashAttention(...)
subgraph = subsch.find(attn_pattern)
subsch.replace(flash_attn, subgraph, name="FA")
```

```
def forward(self, x):
  q, k, v = self.QKV(x)
  attn = self.FA(q, k, v)
  proj = self.OUT(attn)
  output = self.LN(proj)
  return output
```

Figure 4. An example of scheduling computation of a BERT model. — For illustration proposes, we do not show the entire IR of the traced `eff_attn` module, but depict it in an identical `forward` function. `@` denotes dot product. The bias of the new `OUT` module is set as `None`, which has been fused into the `LN` module.

static graph. Third, *just-in-time (JIT)* [4, 56] captures the execution graph in every training iteration, compiles it once, and reuses it in the rest process. Finally, *bytecode-analysis* [44] hooks into the Python frame evaluation to construct the graph from Python bytecode.

In Slapo, we define `.trace(leaves, flatten)` primitive on top of all approaches. This primitive lets developers configure the granularity and the form of a traced graph. Specifically, `leaves` indicate the submodules we will not trace into, and `flatten` indicates whether to flatten a traced static graph. By default, the predefined modules (e.g., `nn.Linear`) in DL frameworks are all leaves, and we create the static graph in a hierarchical way. The specification is then passed to the underlying tracing engine, and the traced module and submodules become static graphs so that compiler-related primitives can be enabled. We show a traced BERT attention module in Fig. 4. In the rest of this subsection, we discuss the scheduling with static graphs.

3.3.1 Partial Computation Scheduling

The computation latency of a module is usually dominated by a part of its computational logic, such as `attn` and `output` in Fig. 4 (highlighted in yellow and green). As a result, it is effective if developers can offload the performance bottleneck logic to efficient kernels or DL compilers. Since most DL models usually have repetitive layers [12, 66], Slapo offers a `.find(regex_or_pattern_fn)` primitive that performs pattern matching algorithm based on subgraph isomorphism [14], with user-specified regular expression or a function with an identical subgraph. This helps find all target subgraphs at once. These subgraphs then can be scheduled in the same fashion with operator fusion, partial computation replacement, activation checkpointing, etc.

Operator Fusion. Operator fusion is an important optimization technique, as it can reduce the data transfer and kernel invocation overheads to improve the latency, throughput, and memory footprint. While existing DL compilers [2, 7, 18, 46] have well-established techniques for conducting operator fusion, distributed training frameworks [51, 55] often cannot leverage them since they do not capture computation graphs and thus cannot apply automatic fusion mechanisms. However, Slapo can partially trace the module to avoid untraceable operators at the same time parallelizing the model, making it effortlessly compatible with existing fusion techniques. As shown in Fig. 4(a), Slapo takes advantage of DL compilers by defining the `.fuse(subgraph, compiler)` primitive, where `compiler` indicates the DL compiler that will be used to generate a fused kernel of the subgraph. We currently support a pattern-based fusion strategy and utilize TorchScript [46] and TorchInductor [45] as DL compilers to enhance kernel performance.

Partial Computation Replacement. In addition to DL compilers, when the subgraph is the performance bottleneck in widely used models, developers may manually implement an efficient kernel and encapsulate it in a module. If this custom kernel achieves better performance than the one generated by a DL compiler, developers are capable of using `.replace(new_mod, subgraph)` primitive to directly replace the corresponding computation logic with the custom one, such as Fig. 4(b). The decision of leveraging a DL compiler or a custom kernel can be made by developers with a one-line change. Developers can also rely on Slapo auto-tuner (§3.4) to realize a better one.

Partial Activation Checkpointing. In addition to enabling activation checkpointing for an entire module, as described in Section 3.2, Slapo offers developers the flexibility to use the `.checkpoint(subgraph)` primitive, allowing checkpointing of specific subgraphs within the computation. This stands in contrast to existing PyTorch-based frameworks, which only support checkpointing at the module level due to their tightly coupled model implementation [51, 55]. By providing this fine-grained control, we address the performance-memory trade-off dilemma, a topic extensively explored in various existing works [8, 22, 26].

3.3.2 Pipeline Partitioning

In §3.2, we demonstrated that Slapo can achieve tensor parallelism using parameter sharding at the module level. However, it is not possible to achieve pipeline parallelism with the same approach, as it requires rewriting the top module, including its `forward` method, to be a sequence of submodules.

SOTA dynamic graph-based DL frameworks support pipeline parallelism in two steps. First, unlike the native runtimes of DL frameworks that use a single process to execute the model graph, pipeline parallelism requires launching one process per pipeline stage. Therefore, DL frameworks with pipeline parallelism support must provide their own runtime. Second, the model must be rewritten to follow a specific API convention. The rewritten implementation consists of a sequence of submodules, with outputs connecting to inputs between two consecutive submodules. This allows the DL framework to assign each module to a stage for execution. However, this process can be tedious for developers to prepare a model for pipeline parallelism [51, 55].

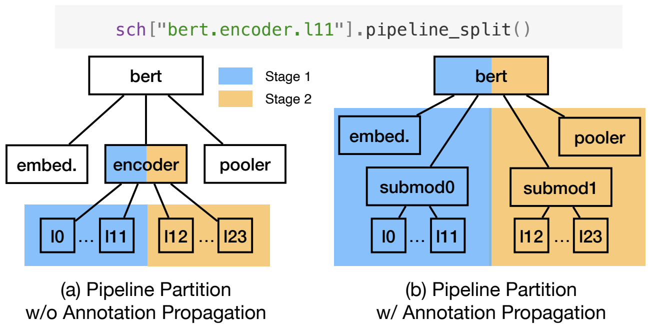
A recent work, PiPPy (Pipeline Parallelism for PyTorch) [23], attempts to address this challenge by tracing the entire model to a static graph and partitioning the graph into a series of modules based on user annotations. However, this approach has limitations, as tracing the entire model can suffer from the limitations of the graph tracer, as discussed in §1. If a part of the model cannot be transformed into a static graph, the entire model cannot be partitioned. In contrast, Slapo allows developers to configure the granularity and the form of the traced graph, meaning that developers can choose to only transform a few top-level modules into a static graph and use the `.pipeline_split()` primitive to annotate the pipeline stage boundaries.

We use the example in Fig. 5 for illustration. To evenly split a BERT model with 24 attention layers in its encoder into two partitions, we can use `.pipeline_split()` primitive[4] to annotate a stage boundary between layer 11 and 12 (0-based) in Fig. 5(a). In this case, only `encoder` module has to be traced, but not its submodules (e.g., `attention`) or siblings (e.g., `embeddings` and `pooler`). We note that the untraceable, complex computation logic usually lies in core building block modules (e.g., `attention`), so limiting the tracing granularity makes our pipeline partitioning more applicable.

However, since Slapo preserves the model structure hierarchy, it is non-trivial to partition a model into a sequence of modules with user-specified pipeline annotations. Specifically, if we simply partition the model based on annotations as Fig. 5(a), we fail to include `embeddings` and `pooler` modules in BERT architecture. To generate the correct partitions, we propose an algorithm that propagates the pipeline annotations from the annotated submodule to the top module, so that all ancestor and descendant modules at all levels can be included. The algorithm is shown as follows.

```
1  def partition(sch, common_parent_sch):
2      seq_modules = partition_by_annotation(sch.module)
3      parent_mod = sch.parent.module
4      inline_and_annotate(seq_modules, parent_mod)
5      if parent_mod != common_parent_sch.module:
6          partition(parent_mod, common_parent_sch)
7
8  sub_schs = sch.get_all_sub_schedules_with_pipeline_split()
9  common_parent_sch = find_common_parent(sub_schs)
10 for sub_sch in sub_schs:
11     partition(sub_sch, common_parent_sch)
12 partition(common_parent_sch, None)
```

[4]Where to insert pipeline splits to achieve optimal throughput is out of scope in this paper, but developers can use auto-tuner in §3.4 for exploration.



```
sch["bert.encoder.l11"].pipeline_split()
```

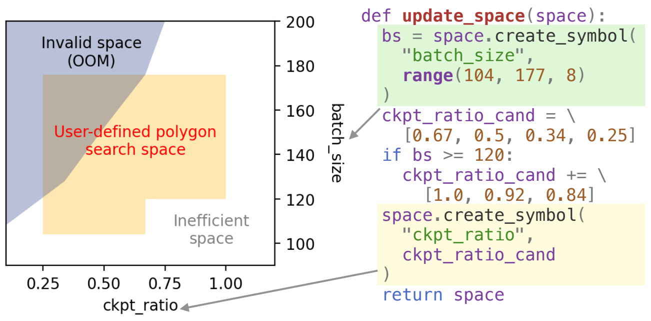
Figure 5. An example of partitioning a BERT model into two pipeline stages.

We first retrieve all the subschedules whose children have pipeline partition annotation (L8). For the example in Fig. 5, only a single subschedule `sch["bert.encoder"]` is returned. We then find the common parent module (L9, `bert` for Fig. 5), and partition each submodule with the annotations (L10-11). After partitioning the current module (L2), we inline the partitioned module sequence and propagate the pipeline split annotations to its parent module (L4) so that the parent module now also has the annotations. We perform this process recursively until the common parent module. At this point, all submodules have been partitioned and inlined, and the common parent module is not partitioned yet but is annotated by its children. Finally, we partition from the common parent module (L12) to the top module to finish the process as depicted in Fig. 5(b).

3.4 Auto-Tuning

Decoupling schedule from model definition enables auto-tuning. The combinations of schedule primitives provided by Slapo can introduce a search space, which includes the decisions of the number of activation checkpoints and pipeline stages, whether to shard a parameter or replace a certain module/subgraph, etc. Consequently, Slapo provides an auto-tuner to explore the best combination given a particular training environment, further reducing the programming burden.

We provide symbolic variable constructions to help developers build a search space with arbitrary numbers of hyperparameters in a schedule. Fig. 6 depicts an example of a search space composed of batch size and the ratio of activation checkpointing. In this example with two tunable



```
def update_space(space):
  bs = space.create_symbol(
    "batch_size",
    range(104, 177, 8)
  )
  ckpt_ratio_cand = \
    [0.67, 0.5, 0.34, 0.25]
  if bs >= 120:
    ckpt_ratio_cand += \
      [1.0, 0.92, 0.84]
  space.create_symbol(
    "ckpt_ratio",
    ckpt_ratio_cand
  )
  return space
```

Figure 6. An example of a user-defined search space.

parameters, we form the candidates of checkpoint ratio based on the batch size, resulting in a polygon search space instead of a rectangle. This allows developers to incorporate domain knowledge to prune inefficient configurations (the white region), and the invalid configurations (the gray region) can be quickly pruned by our auto-tuner.

With the search space constructed, the auto-tuner algorithm iteratively determines the values of all tunable parameters, schedules the model, and launches an evaluation script provided by developers to benchmark the performance and memory footprint. Our auto-tuner leverages exhaustive search by default. Meanwhile, we also provide randomized coordinate descent [40] for users to accelerate the tuning process. We evaluate the efficiency of the search space and the tuning algorithm in §5.4 to show the effectiveness of our auto-tuner.

3.5 Verification

To achieve high usability, Slapo primitives are flexible to schedule modules and computations. However, it is possible for developers to schedule a model incorrectly. For example, the replaced module may require a different data layout but developers do not provide the corresponding layout transformation logic, or developers insert the output aggregation to an invalid point when sharding a parameter tensor.

To guarantee the scheduled model is executable and maintains its numerical correctness, Slapo includes the following verification stages. First, before applying the schedule, we validate the sequence of schedule primitives with a set of predefined rules in each primitive. For example, a `.sync()` primitive must have a corresponding `.shard()` primitive beforehand; primitives for distributed training can only be specified in a distributed environment with multiple devices; primitives that require static graphs must have a corresponding `.trace()` primitive in advance. If any of the rules are violated, the verifier raises an error and stops the rest scheduling process.

Second, we need to assure the numerical correctness of the scheduled model. We provide a `.verify(*schs)` primitive for users to conduct differential testing [37] on different schedules. The verifier generates random inputs and feeds in different scheduled models. This validates both sharded parameter and tensor shapes, as well as confirming consistent outputs with the vanilla model. It works in a distributed environment without altering the model and highlights problematic primitives for improved debuggability.

4 Implementations

We implemented Slapo with ~3K LoC in Python on top of PyTorch [42] to benefit from its dynamic graph and usability. In this section, we highlight some implementation details.

Static Graph Tracing. Our tracer and the static graph are based on `torch.fx` [52], which captures PyTorch models via symbolic tracing and constructs a static graph with a 6-instruction Python-based intermediate representation (IR). Like other DL compilers, `torch.fx` tracer also has unsupported coding styles when capturing the graph, such as certain types of control flow and data structures. It also flattens the IR and discards all the model structural hierarchy. Thus, instead of simply invoking `torch.fx` tracer from the top module, we invoke the tracer module by module and carefully maintain the hierarchy. When a particular module cannot be traced by `torch.fx`, developers can simply disable the corresponding schedule primitives that require a static graph while the rest primitives can still be applied. Since we directly transform the modules and computation graphs and feed the scheduled model to the PyTorch runtime, *no* additional overhead is introduced during execution.

Framework Dialects. Slapo scheduled model is by design compatible with native PyTorch runtime and can be executed on PyTorch directly. To integrate with the dedicated runtime of SOTA distributed systems for certain parallelism (e.g., pipeline), we also implemented two framework dialects for Megatron-LM [55] and DeepSpeed [51]. These systems require either wrapping the model with their custom module or adhering to specific input/output formats. For instance, DeepSpeed [51] pipeline runtime requires a single tuple of inputs and outputs in each pipeline stage, so our DeepSpeed dialect includes automatic logic to (1) unpack the inputs from the previous stage and encapsulate the outputs as a tuple for the next stage, and (2) perform liveness analysis to bypass the tensors that are *not* required by the current stage but are required by subsequent stages. By leveraging the provided dialects, users only need to specify the target framework in Slapo without modifying the model definition.

5 Evaluation

In this section, we evaluate Slapo on different training configurations, in terms of the number of GPUs, number of instances, and parallelism strategies, to demonstrate Slapo is able to align or even outperform existing solutions while preserving usability. Note that Slapo does *not* change the semantics of models and optimizers, so model convergence is *not* affected. We also provide ablation studies to show the effectiveness of the schedule primitives and the auto-tuner.

Setups. All experiments are conducted on Amazon EC2 p3 instances. More specifically, we use p3dn.24xlarge instances with 8×NVIDIA V100 32GB GPUs for single-node evaluations, and use at most 8×p3dn.24xlarge instances for multi-node evaluations. GPUs in these instances are connected via NVLink, which provides 300 GB/s theoretical aggregated GPU interconnect bandwidth, and the inter-node bandwidth is 100 Gbps. The software environment includes CUDA 11.7, PyTorch v2.0.1, Megatron-LM (git-hash 0bb597b), DeepSpeed (v0.9.4), HuggingFace v4.28.1, and NCCL v2.14.3.

Models and Metrics. We apply schedules to a set of popular PyTorch models from HuggingFace Hub [62] and torchvision [36], covering language models and image classification models to demonstrate the generality of Slapo. BERT and RoBERTa are encoder-only Transformer models. GPT and OPT are decoder-only Transformer models. T5 has both encoders and decoders. WideResNet is a convolutional neural network. Detailed model settings are shown in Table 3. Other models like graph neural networks [16, 32] require partitioning the graph structure which is out of our scope. All models in the experiment are trained by AdamW optimizer [35] with mixed precision, and the micro-batch size (i.e., the number of samples per data parallel rank) is selected based on the memory footprint maximizing the system performance. We use the training throughput (the number of total processed samples per second) as our evaluation metric. For each setting, we train the models for tens of steps and take the average throughput after discarding the first few warm-up steps.

Table 3. Models used in the single-node experiments. — # of params shows the model size. MLM = Mask language modeling. CLM = Causal language modeling. Seq2Seq = Sequence-to-Sequence modeling. IC = Image Classification.

Model	Task	# of params (Billion)	Seq Length / Image Size	Precision
BERT [12]	MLM	0.96	512	FP16
RoBERTa [33]	MLM	1.3	512	FP16
GPT [47]	CLM	2.86	1024	FP16
OPT [67]	CLM	2.69	1024	FP16
T5 [48]	Seq2Seq	2.85	1024, 512	FP16
WideResNet [66]	IC	2.4	3×224×224	FP32

5.1 Evaluation on A Single Machine

This subsection evaluates the end-to-end training efficiency on 2, 4, and 8 NVIDIA V100 32GB GPUs in a single p3dn.24xlarge instance to showcase the effectiveness of Slapo.

Systems. We select Megatron-LM v2 [39] as a strong baseline, which is a SOTA system built on top of PyTorch for training large Transformer-based language models on GPUs. Megatron-LM implements its own data loader and optimizer for better training efficiency. In addition, it implements popular Transformer models with tensor parallelism as well as efficient customized CUDA kernels. We also choose DeepSpeed [51] as another baseline. DeepSpeed is a SOTA framework that incorporates ZeRO-powered data parallelism (ZeRO-DP) [50], which applies to arbitrary PyTorch models and is widely used to train large models. We tune both baseline systems by changing the configurations such as batch sizes and activation checkpoints to maximize their performance.

We focus our experiments on frameworks capable of training models that cannot be fit in a single device, and thus traditional data-parallel frameworks [17, 24, 54] are not considered. Additionally, JAX-based frameworks such as Alpa [71] are excluded from the comparison since they are not directly comparable to PyTorch-based frameworks. Alpa also does not exhibit performance advantages when compared to Megatron-LM for the tested models in Table 3, as the model architectures are regular [71]. As Slapo is agnostic to parallelism strategies, we evaluate two configurations for every model to show that Slapo is compatible with the existing distributed training frameworks. Specifically, "Slapo-ZeRO3" schedules models with ZeRO-3 [50] that automatically partitions optimizer states, gradients, and parameters to enable memory-efficient data parallelism; while "Slapo-TP" schedules them to enable tensor parallelism. For each configuration, we auto-tune the checkpointing ratio along with the batch size.

Results. We first compare two baselines, Megatron-LM and DeepSpeed ZeRO-3, in Fig. 7. It is worth noting that Megatron-LM officially only supports three (i.e., BERT, GPT, and T5) out of the six models listed in Table 3. Comparing these three models, we find that no one solution is always superior to the other, highlighting the importance of Slapo which enables developers to easily implement the best parallelism strategies using schedules for different models.

As shown in Fig. 7, Slapo can always perform the best and achieve up to 2.92× speedup compared to the baselines. Slapo-TP achieves throughput gains of 1.02× to 1.46× on 8 GPUs for the models supported by Megatron-LM. Notably, for the BERT model, Slapo-TP can achieve a speedup of up to 1.73×. We employ the tensor parallelism scheme proposed by Megatron-LM to shard both attention and MLP layers, thereby ensuring alignment of multi-device performance. While Megatron-LM implements all the customized kernels within the framework, Slapo captures subgraphs and enables additional optimization opportunities powered by deep learning compilers, thus leading to higher performance. We also note that Slapo-TP does not significantly outperform Megatron-LM on GPT and T5 models. The discrepancy can be attributed to variations in model implementations between Megatron-LM and HuggingFace, which include differences in position embedding, linear bias, layer norm, etc.

The model difference is eliminated when comparing Slapo-ZeRO3 against DeepSpeed since both frameworks run the same HuggingFace models. As illustrated in Fig. 7, Slapo-ZeRO3 consistently outperforms DeepSpeed by a margin

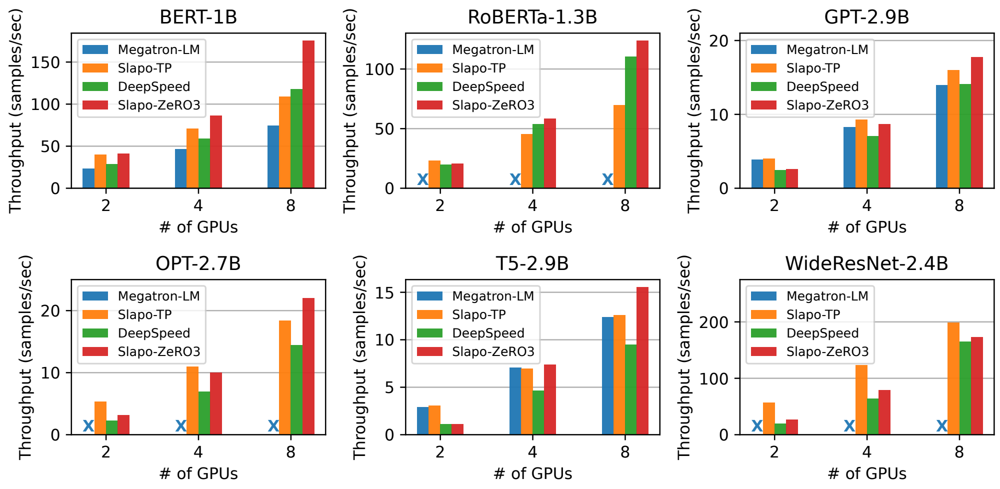


Figure 7. Training throughput on Amazon EC2 p3dn.24xlarge instance with 8 V100 GPUs. "Slapo-TP" uses tensor parallelism to align Megatron-LM. "Slapo-ZeRO3" uses DeepSpeed ZeRO-3 as the parallelism technique. "X" denotes not supported by the framework.

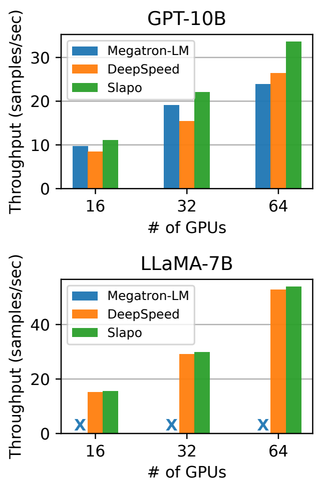


Figure 8. Training throughput of different frameworks on up to 64 V100 GPUs.

of 1.04×-1.64×. The speedup is primarily attributed to the utilization of efficient kernels, which are not included by default in the DeepSpeed training pipeline due to the need for extensive model and parameter rewriting. It is important to note that DeepSpeed supports only fully checkpointed layers implemented in HuggingFace, whereas Slapo can conduct selective checkpointing. By combining this with the auto-tuner, we can consistently achieve higher performance compared to DeepSpeed. We further conduct ablation study in §5.4 to demonstrate the performance improvements of different optimizations.

5.2 Evaluation on Multiple Machines

This subsection presents the results of distributed training performance in multi-machine setups.

Systems and Setups. The testbed in this evaluation is p3dn.24xlarge instances with 8 NVIDIA V100 32 GB GPUs each. We again use DeepSpeed and Megatron-LM v2, as baselines. Following the common practices [39, 50], we use ZeRO-3 for DeepSpeed and set model-parallel and pipeline-parallel size as 8 and 2, respectively, for Megatron-LM. We consider the strong scaling efficiency, in which the global batch size is fixed. Typically, distributed training for large models runs on hundreds to thousands of GPUs and uses global batch sizes up to thousands out of consideration for model quality [5, 10, 39]. We fix the global batch size at 256 for our clusters with up to 64 GPUs, and tune the micro-batch size of each system for the best performance. The evaluation uses a GPT-10B model [47] and a LLaMA-7B model [58]. The input sequence length for the experiments is 1024.

Results. First of all, Fig. 8 reaffirms that no single parallelism strategy consistently performs best for different GPU configurations. In contrast, Slapo consistently outperforms the best available baselines by up to 1.41× when training the GPT-10B model. This is due to Slapo's flexibility, as it is not bound to a specific parallelism strategy or framework. It allows us to choose the most effective strategy and incorporate additional optimizations. By leveraging Slapo, we can readily implement 3D parallelism for emerging models like LLaMA [58, 59], which would require significant engineering efforts to be supported in Megatron-LM. Nevertheless, Slapo achieves a limited speedup over DeepSpeed in the case of LLaMA-7B. According to our profiling, this is because the majority of ZeRO-3 overhead, weight all-gather, is moderate in the 7B-scale model when compared to 3D parallelism.

Table 4. The total lines of code required to implement high-performance schedules (sch) for models in Fig. 7 and Fig. 8.

Model	Sch	Model	Sch	Model	Sch	Model	Sch
BERT	21	GPT	10	T5	11	LLaMA	11
RoBERTa	21	OPT	10	WideResNet	12		

5.3 Usability Study

To demonstrate the improved usability of Slapo, we present the lines of code (LoC) count required to implement high-performance schedules in Table 4. Typical model implementations on the HuggingFace Hub consist of over 1,000 lines of code, making it impractical for developers to directly modify the internal modules scattered in different places to accommodate various hardware environments. Slapo offers a user-friendly interface that allows developers to incorporate the latest optimization techniques without altering the original model definition. In most cases, users can achieve complicated distributed optimizations with about 20 lines of code, significantly reducing the coding burden. Moreover, certain

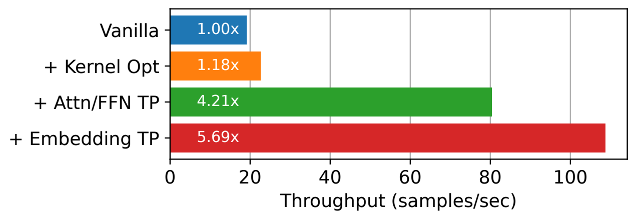


Figure 9. Ablation study with HuggingFace BERT model.

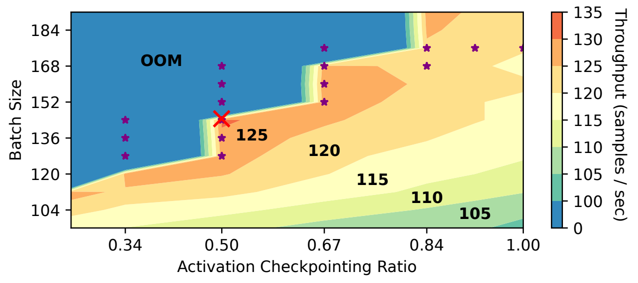


Figure 10. Auto-tuning an OPT-350M model. — Contour lines show the throughput of different combinations of batch size and checkpoint ratio. Throughput 0 means OOM. Purple ★ indicates the explored configurations via coordinate descent; red × depicts the optimal one.

schedules can be shared among models with similar architectures, such as BERT and RoBERTa. Even non-expert users can benefit from our predefined schedule templates to attain high performance. For an illustrative example schedule on the BERT model, please refer to Supplemental Material A.

To further illustrate how Slapo can accommodate various optimizations through the extension of schedule primitives, we analyze its utilization by initial Slapo users. Our investigation assesses several pull requests submitted by different teams, each aiming at adding support for new primitives in our internal repository. We selected several of them and requested detailed reports on the optimization scenarios facilitated by these new primitives, as well as the development efforts required to integrate them. The results are shown in Table 5. The proposed new primitives have been successfully applied to a diverse range of optimization tasks. More importantly, these primitives can be swiftly implemented through the extensible primitive interface in §3.1, often requiring only a single day of development effort. Even the development of an automatic build system for binding CUDA kernels can be accomplished within the same timeframe.

Table 5. The development efforts of introducing new schedule primitives in Slapo. — LoC is the number of lines of code for implementing these optimizations as primitives. The development time does *not* include comprehensive testing on various models.

New primitives	Description	LoC	Approx. Develop Time
`.quantize()`	Replace a module with a predefined quantized module for quantization-aware training	11	1 hour
`.bind()`	Bind a module with a CUDA kernel implementation	95	1 day
`.cudagraphify()`	Use CUDA graph [57] to reduce kernel launch overheads	16	1 hour

5.4 Ablation Study

We design an ablation study for applied optimizations to investigate the performance gain of the schedule.

Schedule Primitives. We start from a vanilla HuggingFace BERT model that is only capable of running on a single device. As evident in Fig. 9, as we progressively apply the schedule primitives to the model, there is a consistent increase in performance. Kernel optimizations, such as Flash Attention [11] and fusing the `Bias-GeLU` kernel, have shown a 1.18× speedup on a single device. By sharding the `attention` and `ffn` modules as discussed in Fig. 3, we can effectively scale out to 8 GPUs, achieving a 4.21× speedup. Moreover, since the word embedding layer contains parameters related to a vocabulary size of over 30K, sharding this parameter results in larger batch sizes, leading to a final speedup of 5.69× and improved scalability when compared to Megatron-LM BERT in Fig. 7.

Auto-Tuning. To showcase the effectiveness of Slapo autotuner within a large search space, we assess the performance of an OPT-350M [67] model using 8 V100 GPUs. We define a search space (including white and yellow regions in Fig. 6) with 91 configurations composed of various batch sizes and activation checkpointing ratios. As shown in Fig. 10, the optimal configuration only checkpoints 50% of the layers with the batch size below the memory threshold, and attains over 30% improved performance compared to the poorest configuration within the search space. With the search space that has already pruned many inefficient configurations, and the coordinate descent algorithm deployed by the Slapo autotuner, only 17 configurations (19% of the total candidates) are explored to identify the optimal configuration with the highest throughput. The entire search process is completed in 20 minutes, as opposed to the 139 minutes an exhaustive search would require, reducing 86% of the search time.

6 Related Work

Schedule Languages and Compilers. Many domain-specific languages (DSLs) leverage the idea of decoupling optimizations from algorithms [2, 6, 7, 19, 28, 49, 63], allowing users to focus on customization and enabling compilers to perform complex optimizations. TVM [7, 15] inherits the decoupling idea from Halide [49] and builds an end-to-end compiler for deep learning inference. Slapo borrows a similar decoupling idea and applies it to the model execution level.

Dynamic Graph Optimizations. Due to the dynamic nature and usability of PyTorch, many frameworks and libraries directly explore optimizations on top of it. ZeRO [50] is a three-stage data parallelism strategy that partitions optimizer states, gradients, and parameters to reduce memory usage, implemented first in DeepSpeed [51] and then adopted by other frameworks [69]. MiCS [68] further improves ZeRO by minimizing the communication scale. Megatron-LM [39, 55] takes a different approach to implementing model parallelism for Transformers, becoming one of the mainstream parallelism libraries. Slapo provides comprehensive primitives to apply optimization techniques in a systematic and productive way. We also use framework dialects described in §4 to train the scheduled models on these frameworks.

Static Graph Optimizations. Some other DL frameworks adopt static graphs so that compiler optimizations can be easily involved. JAX [4] is a popular framework that offers a programming model similar to NumPy, and is powered by XLA [18] as the backend compiler. Accordingly, it is able to achieve 3D parallelism with the corresponding sharding mechanism, GSPMD [64] and GShard [30]. On top of that, Alpa [71] is the first compiler based on JAX and XLA to achieve automatic 3D parallelism. Besides, Unity [60] is another compiler-based distributed training framework that automatically jointly optimizes model parallelism and operator fusion. Their automation mechanisms are orthogonal to Slapo and could inspire Slapo's auto-scheduler in the future.

Moreover, PyTorch 2.0 [43] utilizes `torch.fx` [52] as the IR to capture dynamic graphs and perform optimizations like operator fusion. Nevertheless, it lacks native support for 3D parallelism, partial activation checkpointing, etc., which are crucial for training large models. Slapo takes the heavy lifting work from the users, offering a systematic approach for efficient multi-device model optimization and performance improvement through auto-tuning.

7 Conclusion

In this paper, we propose a schedule language Slapo for progressive optimization of large model training. Slapo decouples model execution from definition, and provides a comprehensive set of schedule primitives for users to efficiently optimize the model execution. Experimental results show Slapo can combine existing optimizations to align or even outperform their performance with minimal programming effort. We plan to implement an auto-scheduler for Slapo to further lower the programming barrier of distributed training, and we believe Slapo can facilitate the rapid prototyping of emerging optimizations for large model training.

Acknowledgments

We would like to thank Jie Wang, anonymous reviewers, and our shepherd, Aditya Kanade, for their insightful feedback. Hongzheng Chen and Zhiru Zhang are supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA and NSF Award #2019306.

A Example Schedule on BERT

The following code snippet shows an example schedule on BERT model using Slapo, where the 21 lines of schedule code are highlighted.

```python
# Import model definition
from transformers import BertLMHeadModel, AutoConfig
config = AutoConfig.from_pretrained("bert-large-uncased")
model = BertLMHeadModel(config)

# Import necessary packages
import slapo
from slapo.pattern import call_module
import torch.nn.functional as F

# Construct schedule for the model
sch = slapo.create_schedule(model)

# Shard embeddings
sch["embeddings.word_embeddings"].sync(mode="fwd_pre",
    sync_op_or_fn=slapo.op.embed_fwd_hook)
sch["embeddings.word_embeddings"].sync(mode="fwd_post",
    sync_op_or_fn=slapo.op.embed_bwd_hook)
for idx in range(config.num_hidden_layers):
    # Shard self attention module
    subsch = sch[f"encoder.layer.{idx}.attention"]
    subsch["self.query"].shard(["weight", "bias"], axis=0)
    subsch["self.key"].shard(["weight", "bias"], axis=0)
    subsch["self.value"].shard(["weight", "bias"], axis=0)
    subsch.sync(mode="bwd_post", sync_op_or_fn="all_reduce")
    subsch["output.dense"].shard("weight", axis=1)
    subsch["output.dense"].sync("fwd_post",
        sync_op_or_fn="all_reduce")
    # Shard MLP module
    subsch = sch[f"encoder.layer.{idx}"]
    subsch["intermediate.dense"].shard(
        ["weight", "bias"], axis=0)
    subsch["intermediate.dense"].sync("bwd_post",
        sync_op_or_fn="all_reduce")
    subsch["output.dense"].shard("weight", axis=1)
    subsch["output.dense"].sync("fwd_post",
        sync_op_or_fn="all_reduce")
    # Decompose linear bias and trace module
    subsch["attention.output.dense"].decompose()
    subsch["output.dense"].decompose()
    subsch.trace(tracer="huggingface", flatten=True)
    # Replace scaled dot product attention
    subgraphs = subsch.find(slapo.pattern.scaled_dot_product)
    subsch.replace(F.scaled_dot_product_attention, subgraphs)
    # Fuse linear bias and gelu
    subgraph = subsch.find(lambda x, bias: F.gelu(bias + x))
    subsch.fuse(subgraph,
        compiler="TorchInductor", name="BiasGeLU")
    # Fuse bias add, layer norm, and residual
    for ln in ["attention.output.LayerNorm",
               "output.LayerNorm"]:
        subgraph = subsch.find(lambda x, bias, residual:
            call_module(ln, F.dropout(bias + x) + residual))
        subsch.fuse(subgraph,
            compiler="TorchInductor", name="LNResidual")
```

B Artifact Appendix

B.1 Abstract

This artifact contains the source code of the Slapo prototype and necessary scripts for configuring the GPU environment and reproducing the experiments in the paper. To make the setup process easier for artifact evaluation, we have packaged our artifact into a docker image, complete with detailed instructions for executing the experiments. A machine equipped with multiple GPUs is required for running the experiments. It takes about 3 hours to run the artifact on an AWS p3dn.24xlarge instance.

B.2 Artifact check-list (meta-information)

- **Model:** BERT [12], RoBERTa [33], GPT [47], OPT [67], T5 [48], and WideResNet [66].
- **Run-time environment:** NVIDIA Container Toolkit.
- **Hardware:** 8×NVIDIA V100 GPUs.
- **Metrics:** Throughput (samples/sec) and lines of code (LoC).
- **Output:** Bar charts and tables.
- **Experiments:** End-to-end throughput evaluation and ablation study on a single machine, and LoC of the schedules.
- **How much disk space required?:** 256 GB.
- **How much time is needed to prepare workflow (approximately)?:** 2 hours for building the docker container.
- **How much time is needed to complete experiments (approximately)?:** 3 hours.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** Both the Slapo artifact and the Slapo language are released under the Apache-2.0 License.
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.10546708

B.3 Description

B.3.1 How to access?

The Slapo framework and the artifact are both available online:

- https://github.com/awslabs/slapo
- https://github.com/chhzh123/slapo-artifact

B.3.2 Hardware dependencies.

All experiments are conducted on Amazon EC2 p3 instances. We use p3dn.24xlarge instances with 8×NVIDIA V100 32GB GPUs for single-node evaluations, and use at most 8×p3dn.24xlarge instances for multi-node evaluations. GPUs in these instances are connected via NVLink, which provides 300 GB/s theoretical aggregated GPU interconnect bandwidth, and the inter-node bandwidth is 100 Gbps. For artifact evaluation, we provide a p3dn.24xlarge instance for reproducing the single-node experiments. Our artifact can be easily customized for various hardware environments. Other NVIDIA GPUs with distinct architectures should also work but will yield different performance results.

B.3.3 Software dependencies.

We provide a docker image for this artifact with NVIDIA GPU support. The software environment includes CUDA 11.7, PyTorch v2.0.1, Megatron-LM (git-hash 0bb597b), DeepSpeed (v0.9.4), HuggingFace v4.28.1, and NCCL v2.14.3.

B.3.4 Datasets.

We use the Wikipedia [21] dataset for evaluating the throughput of Slapo and the baseline systems.

B.4 Installation

To install the artifact, users can clone the repository and build the artifact by themselves:

```
git clone https://github.com/chhzh123/slapo-artifact.git
    --recursive
cd slapo-artifact/3rdparty/slapo/docker
docker build -t slapo -f docker/Dockerfile .
```

Otherwise, users can pull the pre-built docker image from Docker Hub (only compatible with NVIDIA V100 GPUs):

```
docker image pull chhzh123/slapo-ae:latest
docker tag chhzh123/slapo-ae:latest slapo
```

B.5 Experiment workflow

We only provide scripts for reproducing the results of Figure 7, Table 4, and Figure 9, which constitute the main results of our paper. For other experiments, since they may require multiple machines or take excessively long time to run, we do not provide end-to-end evaluation scripts, but users can still find the instructions in our repository.

For a comprehensive guide and the necessary scripts, please refer to the README file included in the artifact. Users can also use the following command to launch a docker image for the experiments:

```
docker run --name slapo --shm-size=150G --gpus all --user
    root -it slapo:latest
```

B.6 Evaluation and expected results

Please refer to the artifact for step-by-step instructions. After executing the experiments, we can obtain:

- Figure 7: The throughput (measured in samples/sec) for Megatron-LM, DeepSpeed, Slapo-TP, and Slapo-ZeRO3 across various models listed in Table 3.
- Table 4: The lines of schedule code of different models.
- Figure 9: The performance breakdown on the BERT model.

B.7 Experiment customization

The Slapo language and the artifact can be further customized for running diverse models and configurations. We provide a benchmarking script that can be easily adapted to accommodate various models, number of GPUs, batch sizes, and sequence lengths. Comprehensive information and guidelines on this can be found in the artifact.

Additionally, we provide detailed documentation and tutorials on using the Slapo language. Please refer to the following link: https://awslabs.github.io/slapo/.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2016.

[2] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2019.

[3] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

[4] 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. http://github.com/google/jax, 2018.

[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. In *Proceedings of the International Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

[6] Hongzheng Chen, Minghua Shen, Nong Xiao, and Yutong Lu. Krill: A compiler and runtime system for concurrent graph processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2021.

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2018.

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

[9] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Proceedings of the International Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

[10] 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. *arXiv preprint arXiv:2204.02311*, 2022.

[11] Tri Dao, Daniel Y Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with io-awareness. *arXiv preprint arXiv:2205.14135*, 2022.

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional Transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[13] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations (ICLR)*, 2021.

[14] David Eppstein. Subgraph isomorphism in planar graphs and related problems. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1995.

[15] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, and Tianqi Chen. TensorIR: An abstraction for automatic tensorized program optimization. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[16] Swapnil Gandhi and Anand Padmanabha Iyer. P3: Distributed deep graph learning at scale. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2021.

[17] Google. Distributed training with tensorflow. https://www.tensorflow.org/guide/distributed_training, 2022.

[18] Google. XLA: Optimizing compiler for machine learning. https://www.tensorflow.org/xla, 2022.

[19] Bastian Hagedorn, Archibald Samuel Elliott, Henrik Barthels, Rastislav Bodik, and Vinod Grover. Fireiron: A data-movement-aware scheduling language for gpus. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2020.

[20] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. GPipe: Efficient training of giant neural networks using pipeline parallelism. In *Proceedings of the International Conference on Neural Information Processing Systems (NeurIPS)*, 2019.

[21] HuggingFace. Wikipedia-en dataset. https://huggingface.co/datasets/wikipedia, 2022.

[22] 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 *Proceedings of Machine Learning and Systems (MLSys)*, 2020.

[23] Ke Wen James Reed, Pavel Belevich. Pippy: Pipeline parallelism for pytorch. https://github.com/pytorch/PiPPy, 2022.

[24] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed dnn training in heterogeneous gpu/cpu clusters. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2020.

[25] Sheng-Chun Kao, Suvinay Subramanian, Gaurav Agrawal, Amir Yazdanbakhsh, and Tushar Krishna. FLAT: An optimized dataflow for mitigating attention bottlenecks. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[26] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. Dynamic tensor rematerialization. In *International Conference on Learning Representations (ICLR)*, 2021.

[27] Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large Transformer models. *arXiv preprint arXiv:2205.05198*, 2022.

[28] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2019.

[29] Benjamin Lefaudeux, Francisco Massa, Diana Liskovich, Wenhan Xiong, Vittorio Caggiano, Sean Naren, Min Xu, Jieru Hu, Marta Tintore, Susan Zhang, Patrick Labatut, and Daniel Haziza. xFormers: A modular and hackable Transformer modelling library. https://github.com/facebookresearch/xformers, 2022.

[30] 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 (ICLR)*, 2021.

[31] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch distributed: Experiences on accelerating data parallel training. *Proc. VLDB Endow.*, 2020.

[32] Tianfeng Liu, Yangrui Chen, Dan Li, Chuan Wu, Yibo Zhu, Jun He, Yanghua Peng, Hongzheng Chen, Hongzhi Chen, and Chuanxiong Guo. BGL: GPU-Efficient GNN training by optimizing graph data I/O and preprocessing. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2023.

[33] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A robustly optimized BERT pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[34] Ze Liu, Han Hu, Yutong Lin, Zhuliang Yao, Zhenda Xie, Yixuan Wei, Jia Ning, Yue Cao, Zheng Zhang, Li Dong, Furu Wei, and Baining Guo. Swin Transformer V2: Scaling up capacity and resolution. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022.

[35] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[36] TorchVision maintainers and contributors. Torchvision: Pytorch's computer vision library. https://github.com/pytorch/vision, 2016.

[37] William M. McKeeman. Differential testing for software. *Digit. Tech. J.*, 1998.

[38] 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 ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[39] 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 (SC)*, 2021.

[40] Yu. Nesterov. Efficiency of coordinate descent methods on huge-scale optimization problems. *SIAM Journal on Optimization*, 2012.

[41] Nvidia. Apex: Tools for easy mixed precision and distributed training in pytorch. https://github.com/NVIDIA/apex, 2022.

[42] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Proceedings of the International Conference on Neural Information Processing Systems (NeurIPS)*, 2019.

[43] PyTorch. PyTorch 2.0. https://pytorch.org/get-started/pytorch-2.0/, 2022.

[44] PyTorch. Torchdynamo overview. https://pytorch.org/docs/master/dynamo/, 2022.

[45] PyTorch. TorchInductor. https://dev-discuss.pytorch.org/t/747, 2022.

[46] PyTorch. TorchScript. https://pytorch.org/docs/stable/jit.html, 2022.

[47] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[48] 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(140):1–67, 2020.

[49] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013.

[50] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.

[51] 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 ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*, 2020.

[52] James Reed, Zachary DeVito, Horace He, Ansley Ussery, and Jason Ansel. torch.fx: Practical program capture and transformation for deep learning in python. *Proceedings of Machine Learning and Systems (MLSys)*, 2022.

[53] Christian Sarofeen, Piotr Bialecki, Kevin Stephano, Jie Jiang, Masaki Kozuki, and Neal Vaidya. Getting started - accelerate your scripts with nvfuser. https://github.com/pytorch/tutorials/blob/0d8c59f/intermediate_source/nvfuser_intro_tutorial.py, 2022.

[54] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[55] 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.

[56] Alex Suhan, Davide Libenzi, Ailing Zhang, Parker Schuh, Brennan Saeta, Jie Young Sohn, and Denys Shabalin. LazyTensor: combining eager execution with domain-specific compilers. *arXiv preprint arXiv:2102.13267*, 2021.

[57] PyTorch Team. Accelerating pytorch with cuda graphs). https://pytorch.org/blog/accelerating-pytorch-with-cuda-graphs/, 2021.

[58] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. LLaMA: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[59] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. LLaMA 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[60] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, Xi Luo, Dheevatsa Mudigere, Jongsoo Park, Misha Smelyanskiy, and Alex Aiken. Unity: Accelerating DNN training through joint optimization of algebraic transformations and parallelization. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2022.

[61] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the International Conference on Neural Information Processing Systems (NeurIPS)*, 2017.

[62] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Huggingface's Transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

[63] Shaojie Xiang, Yi-Hsiang Lai, Yuan Zhou, Hongzheng Chen, Niansong Zhang, Debjit Pal, and Zhiru Zhang. HeteroFlow: An accelerator programming model with decoupled data placement for software-defined fpgas. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2022.

[64] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, Ruoming Pang, Noam Shazeer, Shibo Wang, Tao Wang, Yonghui Wu, and Zhifeng Chen. GSPMD: General and scalable parallelization for ml computation graphs. *arXiv preprint arXiv:2105.04663*, 2021.

[65] Cody Hao Yu, Haozheng Fan, Guangtai Huang, Zhen Jia, Yizhi Liu, Jie Wang, Zach Zheng, Yuan Zhou, Haichen Shen, Junru Shao, Mu Li, and Yida Wang. RAF: Holistic compilation for deep learning model training. *arXiv preprint arXiv:2303.04759*, 2023.

[66] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

[67] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. OPT: Open pre-trained Transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

[68] Zhen Zhang, Shuai Zheng, Yida Wang, Justin Chiu, George Karypis, Trishul Chilimbi, Mu Li, and Xin Jin. MiCS: Near-linear scaling for training gigantic model on public cloud. *Proc. VLDB Endow.*, 2022.

[69] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. Pytorch FSDP: Experiences on scaling fully sharded data parallel. *Proc. VLDB Endow.*, 2023.

[70] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating high-performance tensor programs for deep learning. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2020.

[71] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2022.