Ph.D. DISSERTATION

Systematic Approaches for Efficient Training of Deep Learning Models

효율적인 딥러닝 모델 학습을 위한 시스템적 접근

August 2023

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Taebum Kim

Ph.D. DISSERTATION

Systematic Approaches for Efficient Training of Deep Learning Models

효율적인 딥러닝 모델 학습을 위한 시스템적 접근

August 2023

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Taebum Kim

Systematic Approaches for Efficient Training of Deep Learning Models

효율적인 딥러닝 모델 학습을 위한 시스템적 접근

지도교수 전 병 곤

이 논문을 공학박사 학위논문으로 제출함

2023 년 6 월

서울대학교 대학원

컴퓨터 공학부

김 태 범

김태범의 공학박사 학위논문을 인준함

2023 년 6 월

위 원 장	유 승 주	(인)
부위원장	전 병 곤	(인)
위 원	윤 성 로	(인)
위 원	이 영 기	(인)
위 원	이 윤 성	(인)

Abstract

The growing demand for deep learning (DL) models has created a positive feedback loop with the software systems for DL training. On account of the matured optimizations of such software systems, DL models can be efficiently trained by exploiting the computation resources of DL accelerators. However, several difficulties that hinder the application of such optimizations are still emerging as the structure of models diversifies, and the scale of models increases. Without resolving those difficulties, DL training would yield inefficiencies in practice. In this dissertation, we investigate the reasons for such inefficiencies and design two novel software systems that resolve the inefficiencies.

We first propose Terra, a system that handles the inefficient performance of imperative execution. Terra conducts an imperative-symbolic co-execution that performs the imperative execution of a DL program while delegating the decoupled DL operations to the symbolic execution. Accordingly, Terra can execute any imperative DL program with the optimized performance of the symbolic execution, achieving at most 1.73x speed up compared to the imperative execution. Next, we propose BPipe to resolve the memory inefficiency of pipeline parallelism in large language model training. We introduce a novel pipeline parallelism approach with an activation balancing method. With BPipe, we can train the same model more efficiently, up to 2.17x faster, by making all devices utilize comparable amounts of memory.

Keywords: deep learning framework, large language model training, distributed training

Student Number: 2019-25320

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Efficiency in Deep Learning Model Training

Within the past ten years, deep learning (DL) models have accomplished remarkable achievements in various application domains [49, 30, 31, 114, 20, 25, 9, 76, 96]. Under the advances in DL models, software systems for DL have also flourished, laying the foundation for efficient DL model training [1, 27, 74, 78]. Such systems provide an abstraction layer that can express various DL models and algorithms, along with the optimized runtime that efficiently executes the abstraction on various DL accelerators such as GPU and TPU [41].

However, it might be hard to perform efficient training due to the various constraints when training a DL model in practice. While developing a new DL model, users want to construct an agile development cycle. Accordingly, they prefer a programming interface that helps fast and convenient model development. Unfortunately, convenient usability and fast training speed are difficult to reconcile. In addition, for large language models (LLMs) whose sizes exceed

the memory capacity of a single DL accelerator, we must alleviate high memory pressure to make training can be performed. Yet, it is challenging to relieve the memory pressure without falling into suboptimal training.

In this dissertation, we propose two novel DL training systems: Terra and BPipe. Both systems facilitate efficient training of DL models by resolving different challenges. Terra provides a transparent way to optimize computations while preserving the convenient programming interface, and BPipe provides an efficient method to alleviate the memory pressure of a LLM. In the rest of this chapter, we provide a high-level view of each system with motivations, main ideas, and evaluation results of the two systems.

1.2 Proposed Systems

1.2.1 Imperative-Symbolic Co-Execution of Imperative Deep Learning Programs

The rapid evolution of DL models has been fueled by the support of DL frameworks [1, 27, 78]. Such frameworks provide users with a programming layer to build and execute DL models, commonly adopting Python as their host language. Typically, they execute DL programs with one of the two execution models: imperative or symbolic execution. In the former, the Python interpreter executes a DL program as a normal program, invoking DL operations on-the-fly. The invoked DL operations are executed on a separate DL accelerator asynchronously, and the Python interpreter continues running the program. The dynamic control flows of the DL operations are naturally expressed by the interpretation of the program, and users can utilize any functionalities of Python (e.g., dynamic typing and third-party libraries [10, 29]) while executing DL operations. On the other hand, in the latter model, the Python interpreter

embeds DL operations into a symbolic graph that represents the entire dataflow of a DL model. Thus, users should define their DL programs only with existing symbolic operations that DL frameworks support. In other words, the dynamic control flows of a DL model should be explicitly represented by control flow operations (e.g., `tf.cond` and `tf.while` of TensorFlow). However, the symbolic execution can take advantages of various optimization techniques because the symbolic graph contains a whole computation lineage of a DL model architecture.

Although symbolic execution achieves higher performance compared to imperative execution, imperative execution has been preferred because of its usability. Several systems [11, 83, 84, 27, 36, 67, 107] have been proposed to match the speed of symbolic execution while enjoying the benefit of imperative execution. These systems attempt to generate a symbolic graph that represents an entire imperative program and execute the graph instead of imperatively running the program. Methods for generating the symbolic graph can be broadly classified into two approaches: *single path tracing* and *static compilation*. The former approach generates a symbolic graph by imperatively executing a single iteration of a program and recording the executed DL operations. Systems that adopt the latter approach translate the abstract syntax tree (AST) of a program into a symbolic graph.

Unfortunately, both approaches can correctly handle only a subset of imperative DL programs. For example, dynamic control flows in an imperative program are not captured by the *single path tracing* approach. On the other hand, the *static compilation* approach cannot correctly generate a symbolic graph if a target program contains an AST node that does not have a corresponding symbolic operation such as *try-excepts*, *generators*, *Python object mutations*, and *third-party library calls*. As a result, it is up to the users to

clearly understand the limited usability of these systems.

To overcome the limitations, we propose Terra, an imperative-symbolic co-execution system. While the previous approaches replace the imperative execution with the symbolic execution, Terra maintains the imperative execution to support all Python features where DL operations are delegated to the symbolic execution. Also, Terra generates a symbolic graph by utilizing multiple traces of an imperative program. To be specific, Terra imperatively runs an imperative DL program for several iterations and collects traces, each of which is a linear chain of DL operations sorted by the execution order. The collected traces are merged into a TraceGraph, a directed acyclic graph (DAG) that encapsulates the captured DL operations along with their diverse execution orders. Terra stops collecting traces when the trace of the latest iteration is already embedded in the TraceGraph. To generate an executable symbolic graph from the TraceGraph, Terra adds additional information to the TraceGraph. First of all, Terra annotates the TraceGraph to enable communication between the imperative execution and the symbolic execution. In addition, Terra further analyzes the TraceGraph to insert control flow operations explicitly, so that a symbolic graph can be executed with the DL operations in a certain trace. After generating a symbolic graph from the TraceGraph, Terra starts the co-execution of a skeleton imperative program, in which DL operations are not performed, and the symbolic graph that represents the DL operations. Here, Terra continually checks whether the current trace is being expressed by the TraceGraph. If Terra detects a new trace, Terra discards the symbolic graph and collects more traces by running the original program imperatively. Terra then re-generates the symbolic graph and restarts the co-execution. Consequently, Terra is able to run any imperative DL programs correctly and efficiently even if it contains the Python features that the previous approaches cannot handle.

We have implemented Terra on TensorFlow v2.4.1 and compared Terra with TensorFlow's imperative execution [2] and AutoGraph [67]. Our evaluation shows that Terra can train ten imperative DL programs including convolutional neural networks, transformer-based networks, and generative adversarial networks up to 1.73x faster than the original imperative execution. However, AutoGraph fails to support five programs for three reasons: third-party library call, Python object mutation, and tensor materialization during conversion, which we describe in Chapter 3.

1.2.2 Memory-Balanced Pipeline Parallelism for Training Large Language Models

After the advent of the Transformer architecture [114], there has been a dramatic increase in the size of language models [9, 101, 124, 16, 77, 109, 76]. These models show astonishing results in a wide range of applications by exploiting more than a hundred billion parameters. Such an overwhelming number of parameters incurs high memory pressure, making large language model (LLM) training challenging. When training a model in mixed precision [64] with the Adam [45] optimizer, we need 20 bytes of memory for each model parameter [101]. Hence, training a GPT-3 175B model needs more than 3,000 GiB to store the model parameters and optimizer states. Yet, no GPU exists whose memory capacity satisfies the requirement.

A few methods, such as model parallelism and activation recomputation [28, 46], alleviate the memory pressure to satisfy the requirement. Model parallelism partitions the model parameters and optimizer states across multiple GPUs so that each GPU stores a subset of the model parameters. It is further classified into tensor parallelism [97, 98] and pipeline parallelism [33], where tensor parallelism splits the operations across GPUs and pipeline parallelism splits

the layers across GPUs. On the other hand, activation recomputation releases intermediate activations from memory right after forward computation and recomputes them during backward computation. Since model parallelism and activation recomputation add communication or computation overhead, how we configure them affects training performance significantly. Therefore, finding the configuration that achieves maximum performance and then scaling up training with data parallelism is essential for efficient LLM training.

However, due to its nature, pipeline parallelism could hinder finding the optimal configuration. Unlike tensor parallelism, pipeline parallelism assigns each GPU to handle a separate pipeline stage that computes different layers in a model. Accordingly, each pipeline stage has data dependency on others and results in computation stalls until the required data have arrived, commonly known as a pipeline bubble. To minimize the bubble, a 1F1B (one-forward and one-backward) pipeline schedule [68, 24] splits an input batch into micro-batches and processes forward computation and backward computation alternately. In order to saturate all pipeline stages, earlier stages should reserve more memory for computing more forward micro-batches than later stages. Consequently, a memory imbalance exists across the pipeline stages, and executing the model fails if the earlier stages run out of memory, as illustrated in Figure 3.1.

Our key observation is that the later stages cannot utilize the same amount of GPU memory as the earlier stages require to precompute forward micro-batches. Therefore, if we can exploit the spare memory of later stages as extra memory of the earlier stages, the memory pressure will be relieved with a balanced memory load. In addition, reduced memory pressure allows us to utilize more memory to accelerate training by avoiding redundant recomputations, increasing the micro-batch size, or decreasing the model parallelism degree.

To this end, we propose BPipe, a memory-balanced pipeline parallelism

approach with an activation balancing method to resolve the memory imbalance. While training, BPipe flattens the memory usage of pipeline stages by transferring activations between earlier and later stages. We propose a transfer scheduling algorithm to minimize the number of transfers while preserving the computation correctness. Furthermore, we design a pair-adjacent stage assignment to make transfers not affect the training time. As a result, BPipe achieves a balanced GPU memory usage and facilitates efficient LLM training.

We have implemented BPipe on Megatron-LM [48]. Our evaluation on six HPE Apollo 6500 8-GPU A100 nodes with 800 Gbps cross-node bandwidth shows that BPipe can accelerate training GPT-3 96B and GPT-3 134B models by 1.25x-2.17x by executing more efficient training configurations with fewer recomputations and larger micro-batch sizes.

1.3 Contributions

In this dissertation, we make the following contributions.

- We inspect the inefficiencies of the current systems for DL training. Such inefficiencies cover a wide range: from usability to performance.
- We propose novel systems for DL training, Terra and BPipe, to resolve the inefficiencies.
- We evaluate the two systems and prove that both Terra and BPipe can facilitate more efficient training of DL models.

1.4 Dissertation Overview

The rest of the dissertation is structured as follows. In Chapter 2, we provide background knowledge to help understand Terra and BPipe. Chapter 3

explains Terra, a novel framework for DL programs with imperative-symbolic co-execution. Chapter 4 presents BPipe, a framework for a LLM with a novel pipeline parallelism method. Chapter 5 discusses related works for efficient training of DL models. Finally, we present a future research direction and conclude the dissertation in Chapter 6.

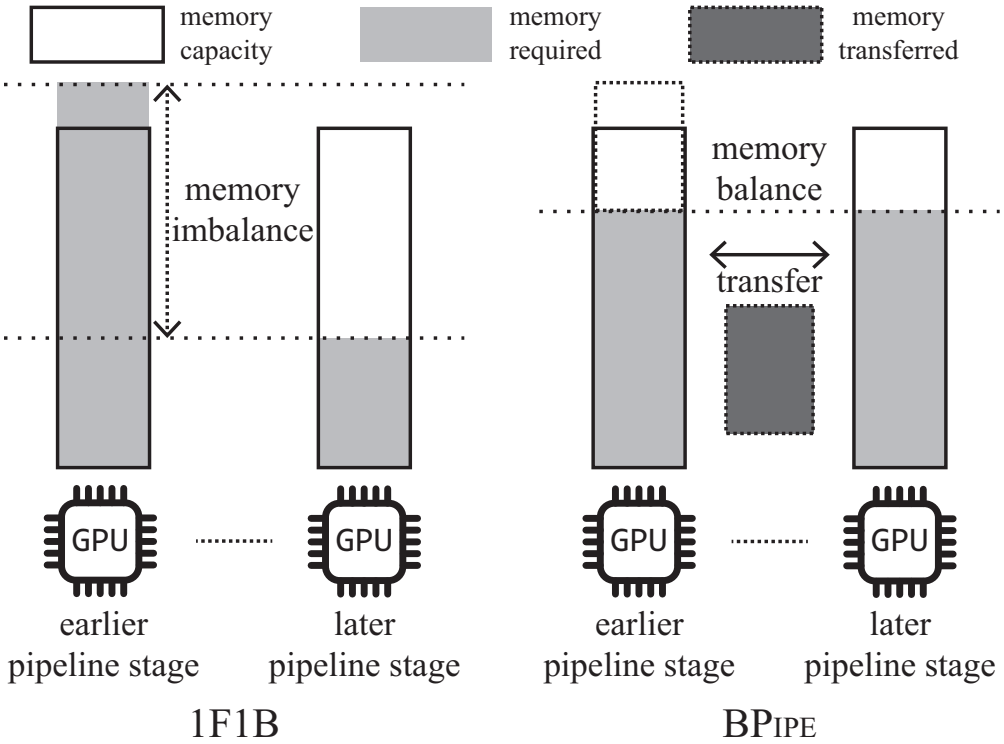


Figure 1.1 An illustration of a memory imbalance and how BPIPE deals with it. With the 1F1B pipeline schedule (left), the memory requirement of an earlier pipeline stage could exceed the memory capacity of a GPU. In that case, a model cannot be executed even if the total memory capacity is sufficient for the memory requirement. BPIPE (right) balances the imbalanced memory requirement by transferring intermediate activations between earlier stages and later stages. Therefore, we can fully utilize the entire memory capacity.

Chapter 2

Background

2.1 Imperative and Symbolic Execution

To train a DL model, users need to prepare a DL program that performs the entire training process. For such a demand, various frameworks have been developed [1, 4, 27, 38, 71, 78, 111, 122] to write a DL program conveniently. Such frameworks commonly adopt Python as their host language and provide users with a programming layer to build and execute DL programs. Typically, they execute DL programs with one of the two execution models: *imperative* or *symbolic* execution.

The **imperative execution** (a.k.a. define-by-run) [71, 78, 111] treats a DL program entirely as a typical Python program. The Python interpreter executes a DL program as a normal program, invoking DL operations on-the-fly. Whenever the Python interpreter encounters a statement that declares a DL operation (e.g., `z = torch.matmul(x, y)`), the interpreter asynchronously invokes a corresponding computation kernel of a DL accelerator (typically GPU

or TPU). The invoked DL operations are executed on the DL accelerator asynchronously, and the Python interpreter continues running the program. Therefore, the imperative execution highly improves the programmability of DL programs because users can fully utilize the convenient language features and rich ecosystem of Python, including built-in functions, dynamic control flows, dynamic typing, and third-party libraries. However, since the imperative execution cannot obtain a whole view of DL model computation, it misses optimization opportunities that the symbolic execution explained below can carry out.

The **symbolic execution** (a.k.a. define-and-run) [38, 1, 4] executes a prebuilt symbolic graph, which represents the entire dataflow of a DL model. The Python interpreter embeds symbolic operations into the symbolic graph and then an optimized graph executor such as TVM [13], TensorRT [75], TensorFlow Runtime [108], and XLA [74] undertakes the actual execution of the symbolic graph. In the symbolic execution, users should express their DL model architectures as symbolic graphs using the three kinds of symbolic operations: DL operations, control flow operations, and auxiliary operations. The DL operations are conventional compute-intensive operations (e.g., matrix multiplication or convolution operation), and the control flow operations determine which DL operations to be executed based on a conditional value within the graph. Finally, to mitigate the limited usability of the symbolic execution, it supports auxiliary operations (e.g., `tf.print` and `tf.py_function` of TensorFlow). After the symbolic graph is constructed, graph optimizations could be applied such as operation fusion [13, 39, 63, 75, 100, 74, 126, 131, 132], parallelized execution [51, 75, 100], device placement [65, 66, 132], layout optimization [39, 62], and memory optimization [3, 32, 34] to accelerate training.

```
def train_step(x):
  y = library_call1()
  loss = model(x, y)
  library_call2(
    loss.numpy())
```

(a) third-party library call & tensor materialization

```
def generator():
  for _ in tf.range(N):
    yield tf.random.normal()

def train_step(x):
  for y in generator():
    x = x + y
  return x
```

(b) dynamic control flow

```
dr = Dropper()
dr.drop_prob = 0.0
def train_step(step, x):
  if step > 100:
    dr.drop_prob = 0.8
  x = tf.nn.dropout(
    x, dr.drop_prob)
  return x
```

(c) Python object mutation

Figure 2.1 Simple examples that the *static compilation* approach cannot deal with. Note that AutoGraph could silently produce an incorrect result in the Python object mutation case.

2.2 Imperative Program with Symbolic Execution

Although the symbolic execution performs faster training, the imperative execution has become mainstream in DL frameworks by virtue of its convenient usability. Therefore, two approaches are proposed to achieve the usability of imperative execution and the performance of symbolic execution simultaneously. They attempt to convert an imperative DL program to a symbolic graph and exploit the symbolic execution with the converted graph. The **single path tracing** approach (e.g., `torch.trace` [83], JAX [27], `torch.fx` [93], and `tf.function` [107]) executes an imperative DL program once and records all DL operations that were executed. A single linear chain of the executed DL operations, which is called a trace, becomes a symbolic graph of the imperative program. The symbolic graph is executed instead of the imperative program for subsequent iterations. Although the *single path tracing* approach looks very simple and intuitive, it is hard to capture the dynamic control flows of the imperative program with this approach. To reflect them correctly, users need to explicitly declare the control flow operations for all dynamic control flows in

their imperative programs, which undermines the programmability of the imperative program. Moreover, Python features that do not have corresponding symbolic operations (e.g., mutation of Python objects, use of third-party libraries) are neither captured by the trace. It can yield an incorrect result since, in the following iterations, a graph executor executes a symbolic graph, which does not contain the Python features.

LazyTensor [102] extends the single path tracing approach by adopting the *lazy evaluation*, in which the Python interpreter and symbolic graph executor run alternately. The Python interpreter executes an imperative DL program as it is and extracts a linear trace of operations. LazyTensor then checks whether the extracted trace is already cached or not. If cached, LazyTensor directly executes the cached graph. If not, it compiles and executes the new graph, then stores the graph for further executions. With the lazy evaluation, LazyTensor could support all Python features without yielding an incorrect result. However, it has an unavoidable performance overhead due to the alternate execution. In other words, the graph executor progresses only when the Python interpreter finishes checking the existence of the cached graph. Similarly, the Python interpreter progresses after finishing the graph execution. Moreover, if an imperative program has a dynamic control flow that is not determined by a tensor value, LazyTensor fails to capture the control flow transparently. In this case, LazyTensor would work inefficiently because the traces could be different for each iteration.

The **static compilation** approach (e.g., TorchScript [84] and JANUS [36]) is proposed to resolve the problem of the *single path tracing* approach. In contrast to the *single path tracing* approach, the *static compilation* approach does not extract a trace to generate a symbolic graph. Instead, it traverses an abstract syntax tree (AST) of the imperative program and directly converts each

AST node to a corresponding symbolic operation. Even if it guarantees the correctness of the program, it fails to convert a program to a symbolic graph if the program contains a Python feature with no corresponding symbolic operation.

We classify the representative failure cases into four categories: *third-party library call*, *tensor materialization during conversion*, *dynamic control flow*, and *Python object mutation*. Figure 2.1 presents simple examples of the cases. The *static compilation* approach cannot convert the third-party library calls of Figure 2.1(a) and fails when it attempts to materialize the tensor data (`loss.numpy()`) during the conversion. Figure 2.1(b) shows the `generator` in Python that the *static compilation* approach cannot convert. For Figure 2.1(c), only JANUS handles it by implementing custom auxiliary operations (i.e., *GetAttr* and *SetAttr* operations) that access the Python heap during the symbolic execution.

To increase the practicality of the previous approaches, AutoGraph [67] combines the *static compilation* approach with the *single path tracing* approach. AutoGraph converts AST nodes that correspond to dynamic control flow such as *if-else*, *for*, and *while* to new AST nodes representing proper control flow operations such as `tf.cond` and `tf.while`. Then, AutoGraph generates a symbolic graph by applying the *single path tracing* approach to the converted AST. Unfortunately, it cannot fully support various kinds of dynamic control flows such as *generator* and *try-except* of Python. Hence, AutoGraph also entails the same correctness problem of the *single path tracing* approach when the imperative DL program employs the features described in Figure 2.1.

When the *static compilation* approach detects unsupported features, it just raises an error and aborts the program execution. To avoid this, users have to write their programs using only the supported features or put in additional efforts such as annotating types and refactoring functions to use tensor objects for

their inputs and outputs. In this regard, AutoGraph [67] and TorchScript [84] provide the official language references [104, 85] that users should be aware of before writing a target imperative program. However, those references usually require expert knowledge to understand, imposing a steep learning curve for new users. For example, the language document of AutoGraph [104] spans roughly twelve pages and divides features that AutoGraph does not support into four categories and ten subcategories. TorchScript provides an official language specification [85] by enumerating supported features over eleven pages.

2.3 Model Parallelism in Large Language Model Training

When training LLMs with limited GPU resources, model parallelism is necessary to split the model into multiple partitions and make each part fit into a single GPU. Depending on how the model is divided, we can classify model parallelism into two categories: tensor parallelism and pipeline parallelism.

Tensor parallelism partitions an operation so that each GPU executes the same operation with partial inputs. However, tensor parallelism degrades the training efficiency with costly synchronization and GPU underutilization. For each partitioned operation, an additional synchronize operation follows to compute the same result as before partitioning. However, it usually needs collective communication such as all-reduce and all-gather, which require a large communication bandwidth. Furthermore, the computation load of a single GPU decreases for the partitioned operation. Then, the GPU could not fully utilize its core. For these limitations, tensor parallelism is known to be practical when used within a node boundary [70]

On the contrary, pipeline parallelism partitions the layers of a model into

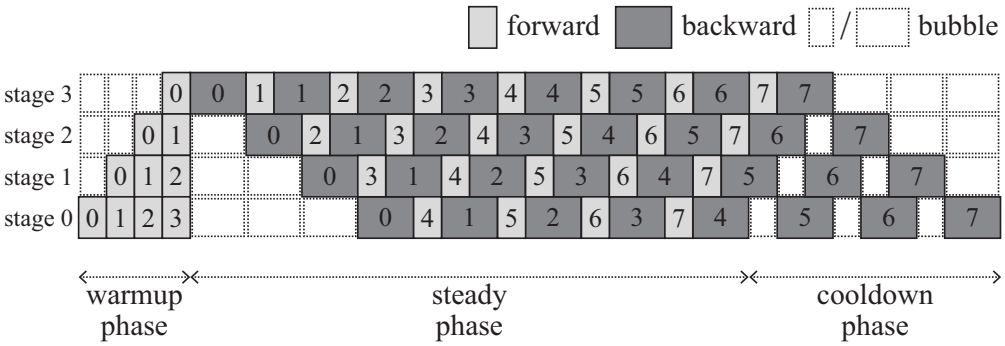


Figure 2.2 An illustration of a 4-way 1F1B pipeline schedule with eight micro-batches. Within the steady phase, forward and backward computation progress alternately. After the cooldown phase, parameters are updated with accumulated gradients of each micro-batch. A number in either forward or backward denotes the micro-batch index.

multiple stages and distributes them across the GPUs. Figure 2.2 illustrates a single training iteration when pipeline parallelism is applied, following the 1F1B pipeline schedule [68, 24]. While training, two consecutive pipeline stages exchange intermediate activations or gradients because each pipeline stage corresponds to different layers. Therefore, point-to-point communication is established between successive pipeline stages. Since the point-to-point communication of pipeline parallelism adds relatively small overhead compared to the synchronization of tensor parallelism, the pipeline parallelism degree can increase along with the model size.

Chapter 3

Imperative-Symbolic Co-Execution of Imperative Deep Learning Programs

3.1 Our Approach: Imperative-Symbolic Co-Execution

To the best of our knowledge, no existing DL framework can completely convert an arbitrary imperative DL program into a symbolic graph. We believe that a one-to-one mapping from all Python features to corresponding symbolic operations should exist to support all imperative programs with the approaches. In other words, building such a mapping is the same as covering all Python syntax with symbolic operations. Moreover, the mapping should be updated as a new feature of Python (e.g., pattern matching of Python 3.10 [82]) is introduced. Eventually, it is equal to building a new Python execution engine for a symbolic graph representing a Python program itself, which requires a tremendous amount of time and effort.

Therefore, we take a different approach that does not replace the entire

imperative execution with the symbolic execution as the previous approaches do. Instead, we let the Python interpreter run an imperative program to support all Python features naturally while separating only DL operations from the imperative execution. As the Python interpreter executes the program except performing DL operations, the decoupled DL operations are executed by a graph executor simultaneously. To enable the co-execution, we generate a symbolic graph representing DL operations that would have been launched by the Python interpreter. While the previous approaches have to build a complete symbolic graph that encapsulates all semantics of the DL program for correctness, we construct a symbolic graph solely based on collected traces of DL operations. Although we do not embed all semantics of the DL program in the symbolic graph, it can handle any DL program by the co-execution of the skeleton program complementing the symbolic execution. Within the skeleton program, DL operations are not performed but all other Python features are preserved as the original imperative program. Executing the symbolic graph in parallel with the skeleton program, we fully achieve the usability of the imperative execution along with the optimized performance of the symbolic execution.

3.2 System Design

Terra is a system that realizes our imperative-symbolic co-execution approach. In this section, we describe how Terra implements the co-execution of a skeleton imperative program and a symbolic graph in detail. There are two requirements to seamlessly maintain the imperative execution along with executing the symbolic graph. First of all, Terra should allow exchanging tensor values between the imperative execution and the symbolic execution when there exists data dependency between each other (e.g., Figure 2.1(a)). Furthermore, the imper-

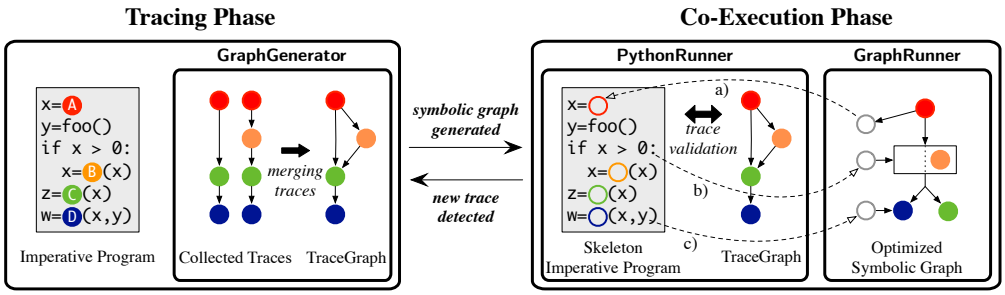


Figure 3.1 An overview of Terra. Each dotted arrow denotes a) the PythonRunner fetches a tensor value from the GraphRunner, b) the PythonRunner informs the GraphRunner of the path that the PythonRunner takes, and c) the PythonRunner feeds an external tensor to the GraphRunner. Rectangle in the optimized symbolic graph denotes the control flow operation.

ative execution should inform the symbolic execution of the correct choice of path to follow because the execution flow of the program is determined by the Python interpreter. To achieve these, Terra implements new symbolic operations for such communication and inserts them into the symbolic graph. In the following, we first describe the entire process of the imperative-symbolic co-execution (§ 3.2.1). Next, we explain how Terra merges collected traces into the TraceGraph and generates a symbolic graph from it in detail (§ 3.2.2).

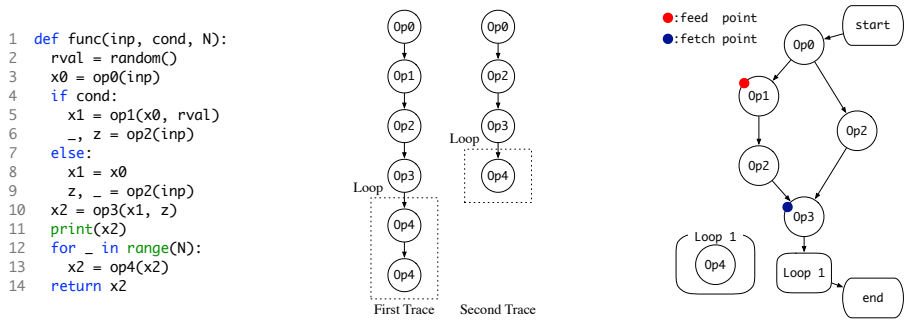
3.2.1 Imperative-Symbolic Co-Execution

The co-execution of Terra consists of the following two phases: the *tracing phase* and the *co-execution phase*. Terra begins execution in the tracing phase, as shown in Figure 3.1. In this phase, the conventional imperative execution is carried out with the given imperative DL program. At the same time, the GraphGenerator collects traces of each iteration. The GraphGenerator incrementally merges the traces into the *TraceGraph*, a directed acyclic graph (DAG)

that encapsulates all the collected traces. Since the number of possible traces during the imperative execution cannot be determined, the GraphGenerator collects traces until the trace of the latest iteration is fully covered in the TraceGraph. In such a case, the GraphGenerator generates a symbolic graph from the TraceGraph.

With the generated symbolic graph, Terra enters the co-execution phase. In this phase, Terra uses the PythonRunner and the GraphRunner. The PythonRunner executes a *skeleton imperative program* that does not launch DL operations anymore. The GraphRunner executes the generated symbolic graph with a separate graph executor. For each DL operation, the PythonRunner skips the actual computation and creates an empty tensor object(s) as an output(s) of the operation. If the PythonRunner has to materialize an empty tensor (e.g., print a loss value), it fetches the actual data from the GraphRunner. Similarly, the GraphRunner might need an external tensor (e.g., an input data, a Python primitive value) from the PythonRunner. Terra implements new symbolic operations to establish the communication. For each communication, a Runner that needs the data from the other waits until the required data becomes ready.

For every iteration in the co-execution phase, the PythonRunner keeps a trace being made by the DL operations in the current iteration. The PythonRunner continuously compares the trace with the TraceGraph to notify the GraphRunner of the current control flow and check the validity of the symbolic graph in the GraphRunner. If the latest DL operation in the trace indicates that the PythonRunner takes a specific path, it informs the GraphRunner of the path with a new symbolic operation, which sets a conditional input of a corresponding control flow operation in the symbolic graph. For example, if the PythonRunner takes the true path of the skeleton imperative program of Figure 3.1 (i.e., `if x > 0:`), the GraphRunner receives such information from the PythonRunner and



(a) Imperative DL Program (b) Collected Traces (c) Merged TraceGraph

Figure 3.2 Illustration of how the TraceGraph is merged from the imperative DL program.

executes the operation of the true path. Furthermore, if the latest DL operation is not expressed in the TraceGraph, Terra considers the current trace as a new trace that the existing symbolic graph cannot handle. Terra then cancels the execution of the GraphRunner and falls back to the tracing phase. Thereafter, the GraphGenerator collects more traces and generates a new symbolic graph covering more traces than before to continue the co-execution.

3.2.2 Symbolic Graph Generation

In this section, we describe how the GraphGenerator merges the collected traces into the TraceGraph and then generates a symbolic graph from the TraceGraph.

TraceGraph Each node of the TraceGraph corresponds to a DL operation, and each edge denotes an execution order between two nodes. For example, if a *Conv2D* operation is followed by another *ReLU* operation in a single trace, the TraceGraph has a directed edge from a *Conv2D* node to a *ReLU* node. For the first trace, the TraceGraph contains a single linear chain of nodes that have two extra nodes; the *start* node and the *end* node. Those nodes do not correspond

to DL operations but for indicating the start point and the end point of the merging.

For subsequent traces, the GraphGenerator attempts to match each operation of the trace with an existing node of the TraceGraph. The GraphGenerator uses a pointer that points to the latest matched node of the TraceGraph, which initially points to the *start* node. For each operation, the GraphGenerator checks whether there exists an equal node among the children of the latest matched node.

When the GraphGenerator checks the equality of two operations while merging multiple traces into a TraceGraph, it compares the type, attributes, and the executed location of each operation. A type of an operation is a kind of the operation, and attributes of operations are information that determines the behavior of the operation. For example, the *MatMul* operation of TensorFlow has '*MatMul*' as its type, and takes *transpose_a* and *transpose_b* as the operation attributes to determine whether the input matrices should be transposed or not. If the GraphGenerator attempts to match the *MatMul* operation whose *transpose_a* is *true* with the *MatMul* operation whose *transpose_a* is *false*, the GraphGenerator fails to match because of the different attributes.

Each executed location of operations stands for the program location in which the operation is actually executed. Since the executed location of the operation is determined at runtime, Terra utilizes a just-in-time (JIT) compilation to evaluate the location. As shown in Figure 3.3, Terra assigns unique *call id*s to every function call and unique *loop id*s to every loop in a given imperative DL program. For each function call, the *call id* of the function is pushed to the *call id stack*, which accumulates the *call id*s. Terra manages the *call id stack* for the entire program execution [1], including the tracing phase and the co-execution

[1]Current implementation of Terra does not consider multi-threading yet.

```
1  def function(x, N):
2    for i in range(N):
3      x = opA(x)
4    return x
```

(a) Original Program

```
1  def function_jit(x, N):
2    try:
3      terra_runtime_info.push_call_id(call_id)
4      try:
5        terra_runtime_info.push_loop_pair(
6            (loop_id, 0))
7        for i in range(N):
8          terra_runtime_info.inc_loop_counter(
9              loop_id)
10         x = opA(x)  # another func call
11     finally:
12       terra_runtime_info.pop_loop_pair()
13     return x
14   finally:
15     terra_runtime_info.pop_call_id()
```

(b) Transformed Program

Figure 3.3 Conceptual illustration of how Terra applies JIT compilation to track a *call id* and a *loop id*

phase. The pushed *call id* is popped when the function is returned. Thus, the *call id stack* contains all information of nested function calls. Similarly, the pair of (*loop id, loop counter=0*) of the loop is pushed to the *loop id stack* for each loop. The *loop counter* is increased for every new iteration of the loop, and the pair of (*loop id, loop counter*) is popped after exiting the loop. As same as the *call id stack*, Terra manages the *loop id stack* for the entire program execution.

After the GraphGenerator finds the child node of the latest matched node which satisfies all criteria, the GraphGenerator updates the latest matched node to that child node, not creating a new node. It then continues merging the next operation of the trace. If all the operations are matched, the GraphGenerator sets the latest matched node to the *end* node, denoting that the current trace is already captured by the TraceGraph.

When the GraphGenerator fails to match the operation with the existing nodes, it denotes that a new trace is detected. A new node is created by creating a new branch from the latest matched node. While expanding the TraceGraph for the new trace, the expanded branch could be merged back into the pre-

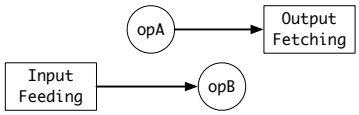
existing branch if there is a node that is not a child of the latest matched node but satisfies all criteria of equality. For example, Figure 3.2(c) depicts the TraceGraph built from the program of Figure 3.2(a), which first took the *true* path (line 5-6) and took the *false* path (line 8-9) at the second execution. From the program, the GraphGenerator collects two different traces as shown in Figure 3.2(b). When the GraphGenerator attempts to merge the second trace into the TraceGraph that contains the first trace, *Op2* of the second trace cannot be matched with *Op1* and cannot be merged back into *Op2* of the first trace because two *Op2*s were executed in different locations. Thus, the node for *Op2* is created in the right branch of Figure 3.2(c), and the branch is merged when the GraphGenerator succeeds to match *Op3*.

As shown in Figure 3.2(c), the GraphGenerator merges the nodes that are executed in the same loop of the program. The GraphGenerator is aware of the loop because it compares the program location where DL operations were executed. It then groups those nodes within an extra loop node and conducts merging the nodes separately. For example, *Loop 1* in Figure 3.2(c) is the loop node for the loop of Figure 3.2(a) (line 12-13). The GraphGenerator merges the second *Op4* of the first trace with the first *Op4* of the first trace because they were executed in the same loop. Also, *Op4* of the second trace is merged to the same node.

Communication between the PythonRunner and the GraphRunner To create the symbolic operations for data communication between the PythonRunner and the GraphRunner, the GraphGenerator captures communication points and annotates such points in the TraceGraph. Those points are classified into *feed* points and *fetch* points. The feed point is where the operation gets an input from the Python interpreter such as training data and Python primitive values.

```
1 x = opA()
2 print(x)
3 k = tf.constant(random.random())
4 # k becomes InputFeeding in the sym. graph
5 y = opB(k)
```

(a) Imperative Program



(b) Symbolic Graph

Figure 3.4 Possible case of deadlock if Terra does not add control dependency between the *Output Fetching* and the *Input Feeding* operations. Note that there is no data dependency between *opA* and *opB* in the symbolic graph.

Similarly, the fetch point is where the Python interpreter needs a value of the DL tensor. For example, *Op1* in Figure 3.2(a) receives *rval* as an input (line 5), and the Python interpreter needs the value of *x2* (line 11) to print it out. The GraphGenerator captures those points and annotates them in the corresponding nodes of the TraceGraph.

Although the PythonRunner executes the skeleton imperative program sequentially, the graph executor of the GraphRunner allows out-of-order execution. Thus, a deadlock could occur if the two Runners conduct the co-execution naively. Suppose that Terra executes the imperative program shown in Figure 3.4(a). Terra generates the symbolic graph from the imperative program as shown in Figure 3.4(b). Since the two operations do not have data and control dependency in the symbolic graph (i.e., *opB* does not consume *opA*'s output), the GraphRunner can freely select the execution order between the operations. If the GraphRunner executes *opB* then *opA*, the deadlock would occur because the PythonRunner should receive the output of *opA* to print its value before it feeds the value k to *opB* in the GraphRunner. To prevent the deadlock, the GraphGenerator adds the control dependencies (defined in TensorFlow) between *Output Fetching* operations that should be executed prior to and an *Input Feeding* operation after generating a symbolic graph. Since the TraceGraph of Terra captures the execution orders between the collected operations, the GraphGen-

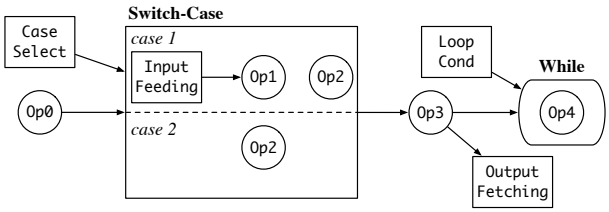
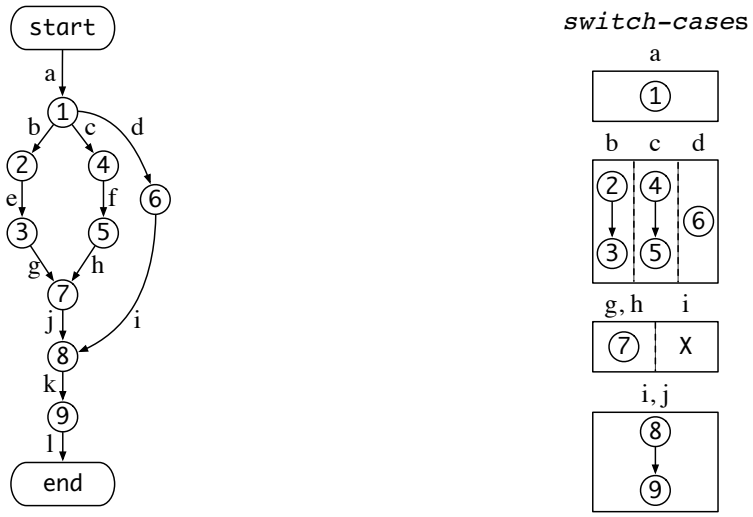


Figure 3.5 Generated symbolic graph from the TraceGraph of Figure 3.2(c)

erator can figure out the control dependencies.

Symbolic graph generation The GraphGenerator converts the nodes in the TraceGraph to the corresponding DL operations and creates additional *Input Feeding* and *Output Fetching* operations to establish data communication during the co-execution. The *Input Feeding* operation corresponds to the feed point of the TraceGraph, enabling the PythonRunner to feed an external tensor to the GraphRunner. Similarly, the *Output Fetching* operation corresponds to the fetch point of the TraceGraph, allowing the PythonRunner to fetch materialized DL tensor from the GraphRunner. As a result, the GraphGenerator represents the entire computation lineage in the single graph with the communication operations. Without those operations, the GraphGenerator should split the symbolic graph into smaller subgraphs at every feed-fetch point, which cannot efficiently apply additional optimizations.

To handle the diverse control flows in the TraceGraph, GraphGenerator utilizes the *Switch-Case* operation (e.g., `tf.case` of TensorFlow), which allows executing only a single *case* that depends on a particular condition. For the conditional input that informs the *Switch-Case* operation of which *case* to execute, the GraphGenerator creates the *Case Select* operation along with the *Switch-Case* operation, as shown in Figure 3.5. When the PythonRunner takes a certain path, it notifies the GraphRunner via the *Case Select* operation.



(a) TraceGraph

(b) The ordered list of *swidth-case*s

Figure 3.6 The result of the case assignment algorithm for the given TraceGraph.

Moreover, the GraphGenerator creates the *While* operation (e.g., `tf.while` of TensorFlow) for a loop node of the TraceGraph. As the *Case Select* operation, the GraphGenerator creates the *Loop Cond* operation along with the *While* operation. The PythonRunner informs the GraphRunner of whether the PythonRunner goes to the next iteration of the loop or exits the loop via the *Loop Cond* operation. As an optimization, the GraphGenerator unrolls the *While* operation if the loop node took the same number of iterations in the collected traces.

We further describe how the GraphGenerator creates the *Switch-Case* operations correctly. GraphGenerator uses *case assignment algorithm* that takes a TraceGraph as an input and returns an ordered list of *switch-case*s. A *switch-case* is a set of (*basic block*, *control edges*) where the *basic block* is a linear chain of nodes, and the *control edges* are the edges that point to the *basic block*. Every non-overlapping linear chain of nodes in the TraceGraph is uniquely assigned

to a *basic block* so that the ordered list of *switch-case*s can cover every trace in the TraceGraph. If there is a loop node in the TraceGraph, the algorithm treats it as a single node because the loop node is converted to the *While* operation in the symbolic graph. For example, from the TraceGraph of Figure 3.6(a), the algorithm returns the ordered list of *switch-case*s of Figure 3.6(b).

Algorithm 1: Terra's case assignment algorithm.

Input: TraceGraph $G = (V \cup \{start, end\}, E)$ where $V = \{v_1, v_2, \ldots, v_n\}$ and $E = \{e_1, e_2, \ldots, e_m\}$

Output: An ordered list of *switch-case*s $S = [s_1, s_2, \ldots, s_p]$

```
1  S ← []
2  next_edges ← { (start, y) ∈ E | y ∈ V }
3  // create a switch_case for each iteration
4  while next_edges ≠ ∅ do
5      switch_case ← ∅
6      new_next_edges ← ∅
7      // create a case of switch_case for each v
8      forall v ∈ { y | (x, y) ∈ next_edges} do
9          control_edges ← edges that point to v among next_edges
10         basic_block ← ∅
11         // no incoming edges to v from E \ next_edges
12         if in-degree(v, E \ next_edges) = 0 then
13             // add v to the basic block
14             basic_block ← basic_block ∪ {v}
15             // expand basic block to contain the linear chain as long as possible
16             while out-degree(v) = 1 and in-degree(next(v)) = 1 and
                 next(v) ≠ end do
17                 v ← next(v)
18                 basic_block ← basic_block ∪ {v}
19             // collect new edges from v
20             new_next_edges ← new_next_edges ∪ { (v, y) ∈ E | y ∈ V }
21         else
22             // keep control_edges for the next iteration of the outer while loop
23             new_next_edges ← new_next_edges ∪ control_edges
24         // update switch_case
25         switch_case ← switch_case ∪ {(basic_block, control_edges)}
26     S.ListAppend(switch_case)
27     next_edges ← new_next_edges
28 return S
```

Algorithm 1 describes how the case assignment algorithm works. The algorithm traverses the given TraceGraph in topological order and makes each *basic block* contain a linear chain of nodes as long as possible. Figure 3.7 shows an example workflow of the algorithm when the TraceGraph of Figure 3.6(a) is an input. At first, the *next_edges* is initialized with {*edge a*} at line 2. Then the algorithm calculates the in-degree of *node 1* from $E \setminus$ *next_edges* at line 12. Since *node 1* has no more incoming edge except for *edge a*, it becomes the first node of *basic_block* at line 14. Then the algorithm attempts to expand *basic_block* as long as possible, but it cannot expand because the out-degree of *node 1* is 3 so that *node 1* is the end of the linear chain (line 16). Thus, the first *switch-case* becomes ({*node 1*}, {*edge a*}) at line 25. At the next iteration, the *next_edges* becomes {*edge b*, *edge c*, *edge d*}, and three *basic block*s are created in the single *switch-case*. Two of them contain the linear chain with two nodes–{*node 2*, *node 3*} and {*node 4*, *node 5*}–and the last *basic block* contains {*node 6*}. When the algorithm processes *edge i* along with {edge g, edge h}, it does not put *node 8* into the *basic block* because the in-degree of *node 8* is not zero (line 12) due to *edge j*. Thus, the *basic block* becomes an empty set. Finally, the algorithm returns the ordered list of *switch-case*s after creating the *basic block* with *node 8* and *node 9*.

As shown in Figure 3.7, each *switch-case* within the result of the case assignment algorithm becomes the *Switch-Case* operation in the symbolic graph. If a *switch-case* contains only a single *basic block*, the GraphGenerator does not create a redundant *Switch-Case* operation. For each *Switch-Case* operation, the GraphGenerator creates the *Case Select* operation. During the co-execution, the PythonRunner informs the GraphRunner of the control edge taken via the *Case Select* operation. For example, if the PythonRunner follows *edge c* of Figure 3.7, the GraphRunner executes *case 2* of the first *Switch-Case* operation.

Now we describe the formal definitions and the proof of the correctness of the algorithm.

Definition 1. A **TraceGraph** $G = (V \cup \{start, end\}, E)$ is a directed acyclic graph (DAG) where V is set of nodes ($V = \{v_1, \ldots, v_n\}$) and E is set of directed edges ($E = \{e_1, \ldots, e_m\}$) that connect the nodes. The TraceGraph has two extra nodes: the *start* node and the *end* node. The *start* node is a unique source node (i.e., in-degree of the *start* node is 0) and the *end* node is a unique sink node (i.e., out-degree of the *end* node is 0) of the TraceGraph.

Definition 2. A **linear chain** is an ordered set of nodes $L = \{v_1, \ldots, v_l\} \subseteq V$ such that for all $2 \leq i \leq l$, $(v_{i-1}, v_i) \in E$, the in-degrees of all nodes are 1 except v_1, and the out-degrees of all nodes are 1 except v_l. Also, the ordered set of edges in the linear chain, $I(L) = \{\ (v_{i-1}, v_i) \mid 2 \leq i \leq l\ \}$, is called **in-chain edges**.

Definition 3. A **case** c is a pair of (**basic block**, **control edges**) $= (L_c, E_c)$ where $L_c = \{v_1, \ldots, v_l\}$ is a *linear chain* and E_c is a subset of E with the edges that point to v_1 of L_c. In addition, a **switch-case** s is a set of *case*s that satisfies the following condition:

$$\forall\ c_1, c_2 \in s \text{ such that } c_1 \neq c_2, L_{c_1} \cap L_{c_2} = \emptyset \text{ and } E_{c_1} \cap E_{c_2} = \emptyset.$$

In other words, different *case*s are mutually exclusive.

Definition 4. A **trace** $t = (V_t \cup \{start, end\}, E_t)$ is a DAG that satisfies the following conditions:

1. $V_t = \{v_1, \ldots, v_{k-1}\} \subseteq V$ and $E_t = \{e_1, \ldots, e_k\} \subseteq E$
2. $\forall 1 \leq i \leq k, e_i = (v_{i-1}, v_i)$ where $v_0 = start, v_k = end$

Moreover, the **operation nodes** $Op(t)$ of the *trace* t is V_t, and the **path** $Path(t)$ of the *trace* t is E_t.

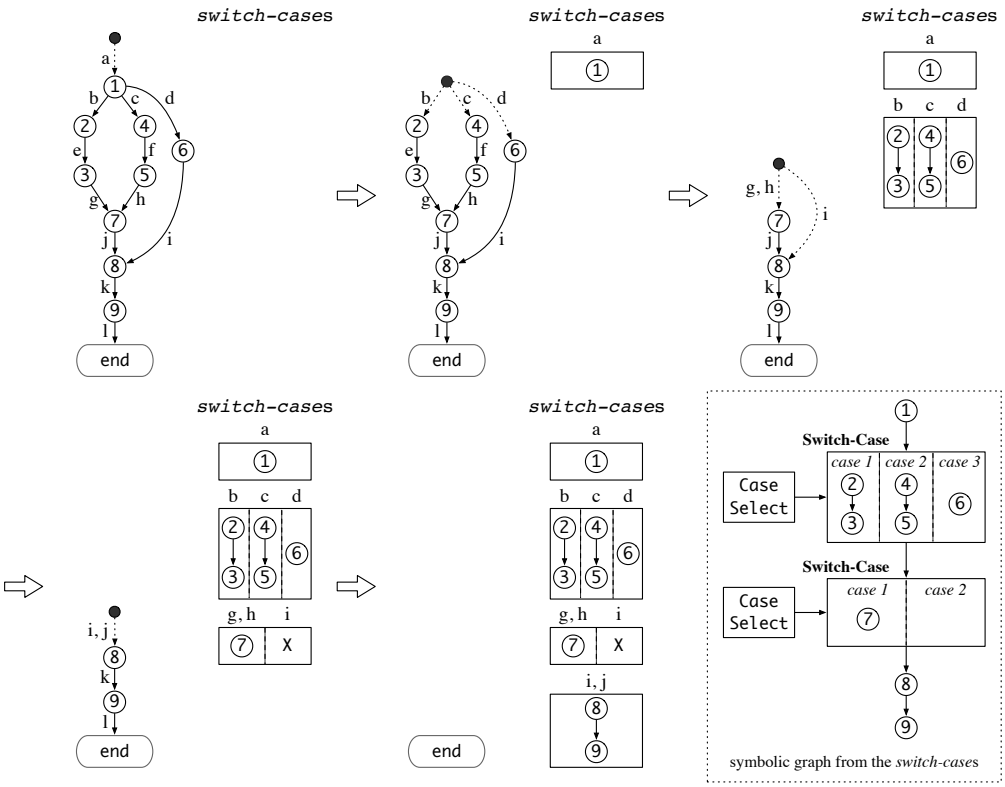


Figure 3.7 Example workflow of how the case assignment algorithm works, and how the symbolic graph is generated from the ordered list of *switch-case*s. The dotted arrows from a black circle denote the *next_ edges* variable of Algorithm 1. All processed nodes and edges are assigned to an appropriate *switch-case*, where each rectangle of the *switch-case*s denotes the *basic block*. `X` denotes that no node exists in the *basic block*.

Definition 5. An ordered list $S = [s_1, \ldots, s_p]$ of *switch-case*s covers a *trace* t if the following conditions hold.

1. For all s_i, there exists a unique *case* $c_i = (L_{c_i}, E_{c_i}) \in s_i$ with a unique edge d_i such that $\{d_i\} = E_{c_i} \cap Path(t)$. The other *case*s do not have corresponding *control edges* for $Path(t)$.

2. The operations in the trace are represented as the linear chains of the c_i's,

and all edges in the trace are the union of *in-chain edges*, the d_i's, and e_k. That is,

$$Op(t) = \bigcup_{i=1}^{p} L_{c_i} \text{ and } Path(t) = \left[\bigcup_{i=1}^{p}(\{d_i\} \cup I(L_{c_i}))\right] \cup \{e_k\}.$$

Note that e_k is the edge that points to the *end* node.

Definition 6. A graph $G_s = (V_s \cup \{start, end\}, E_s)$ is a **sub-TraceGraph** with V_s if

1. G_s is a TraceGraph and $V_s \subseteq V$.
2. $E_s = E_1 \cup E_2$ where

$$E_1 = \{\ (u, v) \in E \mid u \in V_s \cup \{start\},\ v \in V_s \cup \{end\}\ \}$$
$$E_2 = \{\ (u, end) \mid (u, v) \in E,\ u \in V_s \cup \{start\}\ ,\ v \notin V_s \cup \{end\}\ \}.$$

To be specific, E_1 denotes all the edges between the nodes within $V_s \cup \{start, end\}$. Furthermore, for all edges whose source node u is in $V_s \cup \{start\}$ and destination node v is not in $V_s \cup \{end\}$, the sub-TraceGraph changes the destination node of such edges to the *end* node because the *end* node should be a unique sink node of the TraceGraph. Then, E_2 denotes the changed edges. We define the sub-TraceGraph to use in the proof of the following theorem.

Theorem 1. Algorithm 1 generates an ordered list S of *switch-case*s that covers every *trace* in the TraceGraph $G = (V \cup \{start, end\}, E)$.

Proof of Theorem 1. To prove the theorem, we define the following auxiliary variables:

- *processed nodes* $V_p = \bigcup_{s \in S} \bigcup_{c \in s} \{\ v \in L_c \mid c = (L_c, E_c)\ \}$
- *connecting edges* $C = \{\ (u, v) \in E \mid u \in V_p \cup \{start\}\ , v \notin V_p \cup \{end\}\}$

- *processed TraceGraph* $G_p = (\ V_p \cup \{\ start, end\ \}, E_p\ )$

First, the *processed nodes* is a set of all nodes in S, which is the same as a set of the nodes that the algorithm visited. The *connecting edges* is a set of edges where the source node of each edge is in $V_p \cup \{start\}$ and the destination node of each edge is not in $V_p \cup \{end\}$. Finally, the *processed TraceGraph* represents the TraceGraph with the *processed nodes*. It has E_p such that

$$\begin{aligned} E_p =& \{\ (u, v) \in E \mid u \in V_p \cup \{start\},\ v \in V_p \cup \{end\}\ \} \cup \\ & \{(u, end) \mid (u, v) \in C\}. \end{aligned}$$

Then, we use the following loop invariants prior to every iteration of the loop at line 4.

1. The *next_ edges* variable of the algorithm is identical to the *connecting edges* C.

2. The *processed TraceGraph* G_p is a *sub-TraceGraph* with V_p.

3. The variable S is an ordered list of *switch-case*s, and it covers every *trace* in G_p.

At the beginning of the loop, the three loop invariants hold with $S = []$, $V_p = \emptyset, C = \{\ (u, v) \in E \mid u = start, v \neq end\ \}$, and $E_p = \{(start, end)\}$.

Next, we prove that the loop invariants are maintained after each iteration. For each v at line 8, the variable *basic_ block* is a *linear chain* collected throughout the while loop of line 16 or an empty set. If the *basic_ block* contains a *linear chain*, V_p adds all the nodes within the *linear chain*. Then the *next_ edges* becomes the new edges that point to the nodes which are not in $V_p \cup \{end\}$ (line 20). If the *basic_ block* is an empty set, it denotes that v is not added to V_p. Then, the *next_ edges* becomes the *control_ edges*, where all the

edges are pointing to v (line 23). Thus, the first loop invariant holds. Moreover, the second loop invariant holds because the *linear chain*s of each iteration extend V_p with corresponding nodes while including *in-chain edges* and updating the *connecting edges*. Subsequently, the third loop invariant holds because each *control edges* is assigned to the specific *case* with the corresponding *linear chain* (line 25).

Finally, we prove the theorem by showing the following propositions are true.

1. For each iteration, $|V_p|$ strictly increases.
2. After the termination of the outermost while loop, $V_p = V$ and $G_p = G$.

The first proposition shows that the outermost loop eventually finishes, and the second proposition shows that the variable S covers every *trace* in G. First of all, for an iteration, let $N = \{v_1, v_2, \ldots, v_{|N|}\}$ from $\{\, y \mid (x, y) \in \text{next\_edges}\}$ at line 8. Then, suppose that $|V_p|$ is not increased, which denotes

$$\forall v_i \in N, \text{in-degree}(v_i, E \setminus \text{next\_edges}) \neq 0$$

at line 12. In other words, it implies that

$$\forall\ v_i, \exists\ v_j \in N \text{ such that } v_j \neq v_i, v_j \sim v_i$$

where $x \sim y$ indicates that for $x \in V$ and $y \in V$, there exists a path from x to y in G. Without loss of generality, assume that $v_2 \sim v_1$. Then, there should exist j such that $3 \leq j \leq |N|$ and $v_j \sim v_2$. However, this requires a cycle in G in the end, which contradicts to the assumption: G is a DAG. Thus, $|V_p|$ strictly increases throughout the iterations.

Now suppose that $V_p \nsubseteq V$ after the termination of the outermost loop. Then, there exists $v \in V \setminus V_p$. However, this contradicts to the termination condition

of the loop because there exists a path from *start* to v by the definition of the TraceGraph. Thus, $V_p = V$ and $G_p = G$ after the termination of the outermost while loop. □

3.3 Evaluation

In this section, we evaluate Terra in the following two aspects:

- Can Terra exploit symbolic execution from imperative DL programs that AutoGraph, the static-compilation-and-tracing approach, cannot? (§ 3.3.3)
- How much does Terra speed up imperative DL programs? (§ 3.3.4)

3.3.1 Implementation Detail

Frameworks We use TensorFlow [1] v2.4.1 as our baseline DL framework. We have built Terra on TensorFlow v2.4.1, and our approach is applicable to other DL frameworks if they support both imperative and symbolic execution (e.g., MXNet [12] and PyTorch [78]).

We modified the imperative execution model of TensorFlow for both GraphGenerator and PythonRunner. When the Python interpreter executes DL operations imperatively in the tracing phase, the interpreter makes the GraphGenerator record each operation as a symbolic representation, which is a *NodeDef* of TensorFlow. During the co-execution phase, functions that trigger an actual computation of a DL operation are modified to perform validating the symbolic graph and creating an empty tensor object. To annotate feed points, we modified `FuncGraph.capture` to capture all external tensors. Similarly, we modified `EagerTensor.numpy` to annotate fetch points.

All evaluated imperative DL programs are implemented with the imperative API of TensorFlow, which has become the standard interface since TensorFlow

```
1  import tensorflow as tf
2
3  for inputs, labels in train_data_loader:
4    with tf.TerraGradientTape() as tape:
5      logits = model(inputs)
6      loss = loss_fn(logits, labels)
7    grads = tape.gradient(
8        loss, model.trainable_variables)
9    optimizer.apply_gradients(
10       zip(grads, model.trainable_variables))
```

Figure 3.8 Programming interface of Terra

v2.0. We compare Terra with TensorFlow imperative execution and with AutoGraph [67], a state-of-the-art system that combines the *static compilation* approach with the *single path tracing* approach. We compile a single training step function of each imperative DL program (i.e., `@tf.function(autograph=True)`).

Usability Figure 3.8 shows an example code of using Terra to speed up the training process. All the programmers have to do is just to modify a single line of code in their imperative DL program: changing from `tf.GradientTape` to `tf.TerraGradientTape` at line 4. Since all imperative TensorFlow programs must use `GradientTape` to train DL models, Terra is applicable to all imperative programs transparently without programmers' extra burden. Terra generates the symbolic graph from the DL operations within the `TerraGradientTape` context and the gradient computations (`tape.gradient`).

Furthermore, since Terra utilizes the Python Interpreter throughout the execution, it provides a transparent debuggability as the imperative execution does. Terra can print out the same traceback as executing as the imperative execution while AutoGraph cannot.

Fallback handling When the PythonRunner detects a new trace in the co-execution phase, Terra cancels the execution of the GraphRunner. Then, the

PythonRunner executes all the DL operations that have been matched within the current iteration to make the program state as if it were being performed imperatively from the beginning. While executing the matched operations, some of them could be executed twice if the GraphRunner already executed the operations. This can be a problem for stateful operations, which hold and change the program state such as I/O operations and communication operations. To prevent this problem, stateful operations are not recorded by the GraphGenerator so that those operations are not included in the symbolic graph. Any inputs and outputs of the stateful operations are connected with the symbolic graph through the *Input Feeding* and the *Output Fetching* operations.

We exceptionally allow the GraphGenerator to record and generate stateful operations that are related to variables (both trainable and non-trainable) of a DL model such as the *ReadVariableOp* operation and the *AssignVariableOp* operation of TensorFlow to optimize performance. To ensure correct execution, the GraphGenerator inserts control dependencies between those operations (e.g., ensuring read after write) automatically while generating the symbolic graph. Furthermore, for the variables whose update operations are generated in the symbolic graph (e.g., updating moving averages of batch normalization), the PythonRunner makes a checkpoint of those variables at the beginning of each iteration. Whenever the GraphRunner's execution is canceled, the PythonRunner restores such variables from the checkpoint and executes the operations that the PythonRunner has succeeded to match.

3.3.2 Experiment Setup

Environments We conduct all the experiments on a single machine that is equipped with 8-core AMD Ryzen 7 2700X @ 3.7GHz and an NVIDIA TITAN Xp GPU. We use Ubuntu 18.04, CUDA 11.0, cuDNN 8.0, and Python 3.8.8.

Table 3.1 The programs that AutoGraph fails to execute and the reason for the failures. Note that Terra can execute all of them.

Program	Reason of the failure
DropBlock [21]	Python object mutation
MusicTransformer [44]	Python object mutation
SDPoint [50]	Python object mutation
BERT-CLS [52]	third-party library call
FasterRCNN [116]	tensor materialization during conversion

Imperative DL Programs For the experiments, we use ten imperative DL programs collected from open-source GitHub repositories: DropBlock [21], BERT-Q&A [22], MusicTransformer [44], SDPoint [50], BERT-CLS [52], GPT2 [99], DCGAN [105], ResNet50 [106], FasterRCNN [116], and YOLOv3 [125].

3.3.3 Imperative Program Coverage

Terra handles all the benchmark programs successfully with the imperative-symbolic co-execution. However, since AutoGraph does not support the entire set of Python features, it fails to execute five out of ten programs.

Figure 3.9 shows the codes that AutoGraph fails to convert. First, DropBlock [21] keeps `keep_prob` in the class object and alters it during training. However, AutoGraph cannot detect the mutation throughout the training. Similarly, AutoGraph cannot capture the object mutations of both MusicTransformer [44] and SDPoint [50]. For MusicTransformer, the object mutation is not related to the algorithmic characteristic of the model but the programming style of the user. It wraps the entire training process in a single *trainer* class, which is a common design pattern for implementing a program that trains a DL model [23]. The `_train_step` method calculates the loss value of the model for each training step, and it writes the value to the `loss_value` attribute

```
class DropBlock2D(tf.keras.layers.Layer):
  def set_keep_prob(self, keep_prob=None):
    if keep_prob is not None:
      self.keep_prob = keep_prob
    w, h = tf.cast(self.w, tf.float32), \
        tf.cast(self.h, tf.float32)
    self.gamma = (1. - self.keep_prob) * (w * h) \
        ...
 def call(self, x, training=False):
    ...
    mask = _bernoulli(sampling_mask_shape, self.gamma)
    ...
```

(a) DropBlock

```
class MusicTransformerDecoder(tf.keras.models.Model):
  def _train_step(self, inp_tar, out_tar,
                  lookup_mask, training):
    predictions = self.call(inp_tar, lookup_mask,
                            training)
    self.loss_value = self.loss(out_tar, predictions)
    ...

  def train_on_batch(self, ...):
    # not a conversion scope
    ...
    loss = tf.reduce_mean(self.loss_value)
```

(b) MusicTransformer

```
class SDResNet(tf.keras.models.Model):
  def stochastic_downsampling(self, blockID, ratio):
    downsampling_ratio = ratio is None and 0.5 or ratio
    for l in self.layers:
      if isinstance(l, _ConvBlock):
        if l.blockID == blockID:
          l.downsampling_ratio = downsampling_ratio
        else:
          l.downsampling_ratio = 1.
```

(c) SDPoint

```
from sklearn.metrics import f1_score
class SparseF1Score(object):
  def __call__(self, y_true, y_predict):
    y_true = tf.reshape(
        tf.constant(y_true), [-1]).numpy()
    y_predict = tf.reshape(
        tf.argmax(y_predict, -1), [-1]).numpy()
    f1 = f1_score(y_true, y_predict,
        average=self.average)
    return f1
```

(d) BERT-CLS

```
def calc_batch_padded_shape(meta):
  return tf.cast(tf.reduce_max(
      meta[:, 6:8], axis=0), tf.int32).numpy()
```

(e) FasterRCNN

Figure 3.9 Code snippets that AutoGraph fails to convert correctly.

(line 6 of Figure 3.9(b)). However, AutoGraph cannot write the new loss value to the attribute because it does not access the Python heap while carrying out the symbolic execution. Thus, when the Python interpreter attempts to read `loss_value` (line 12), it fails to read the updated loss. Terra correctly captures those mutations because the PythonRunner accesses the Python heap and updates the objects in the *co-execution* phase. BERT-CLS [53] and Faster-RCNN [116] show the example cases of *tensor materialization during conversion*. For both cases, AutoGraph fails to generate a graph because they cannot evaluate the exact value of the tensors while generating the symbolic graph. Moreover, BERT-CLS should evaluate the tensor values to calculate the target metric via a third-party library [10], which AutoGraph does not support. However, Terra is not affected by such cases because the GraphGenerator collects traces while Terra carrying out the imperative execution in the tracing phase. Then in the co-execution phase, the PythonRunner materializes those tensors via the *Output Fetching* operations of the symbolic graph.

According to the AutoGraph language document [104], more failures could exist if an imperative DL program contains unsupported Python features such as the use of Python *generator*, *try-except*, and *None* type values. To resolve all the limitations in AutoGraph, new functions to handle each failure should be implemented and it requires a huge engineering effort. Terra simply avoids the conversion-related problems by the imperative-symbolic co-execution.

3.3.4 Training Throughput

Figure 3.10 presents the training speed-up of Terra compared to TensorFlow imperative execution. For all programs, we measure the average training throughputs from 100 to 200 steps, and each experiment is conducted ten times. Terra achieves higher performance than the imperative execution for every program.

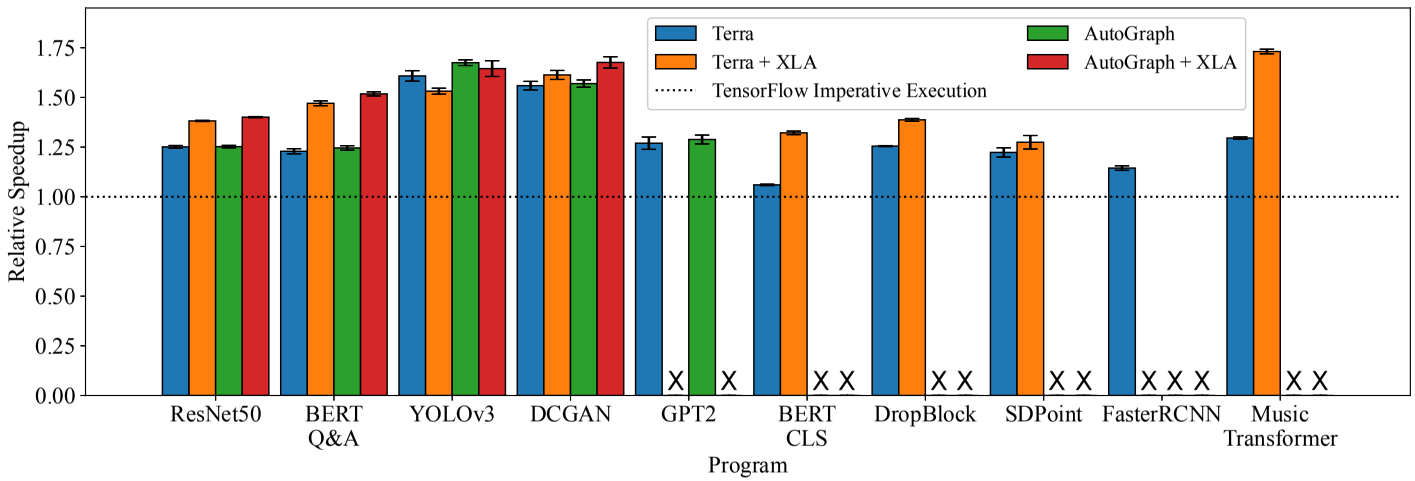


Figure 3.10 The training speed-up results of Terra, AutoGraph, and when applying XLA [74] to both systems relative to TensorFlow imperative execution. The dotted line presents the training throughput of the imperative execution. Note that Terra and AutoGraph share the same upper bound of performance improvement from the symbolic execution of TensorFlow.

To estimate whether Terra fully achieves the optimized performance of symbolic execution, we also compare the performance of Terra with AutoGraph, which shares the same graph executor of TensorFlow with Terra. AutoGraph closely follows the performance of the symbolic execution because it totally replaces the imperative execution with the symbolic execution. For the five programs that AutoGraph can execute, the performance improvements of Terra are on par with AutoGraph, which shows that Terra highly achieves the symbolic execution's optimized performance. Experiment settings such as batch size and the dataset are included in Appendix E.

Since Terra generates a symbolic graph and utilizes the symbolic execution, we evaluate the performance of Terra by applying XLA [74] as shown in Figure 3.10. Compared to the imperative execution, Terra improves the performance of seven programs by up to 1.73x when applying XLA. XLA is not applicable to GPT2 and FasterRCNN due to the dynamic shape of the input data. For each training iteration, the shapes of input data to the models can change. However, XLA cannot efficiently handle dynamic shapes because it assumes static shapes. For YOLOv3, we profile the execution and find that the current XLA fails to efficiently cluster operations for YOLOv3. To be more specific, YOLOv3 includes some DL operations such as *ResizeNearestNeighbor* and *Where*, which are not supported by XLA. Thus, XLA cannot efficiently fuse DL operations. In addition, we observe that Terra's performance decreases more than that of AutoGraph for YOLOv3 because the schedules of some *Output Fetching* operations are reordered because of XLA kernels, causing a longer stall in the PythonRunner. However, this problem can be addressed by extending XLA to support our custom operations.

We profile the execution of both PythonRunner and GraphRunner to analyze the performance of Terra. We focus on the performance analysis in the *co-*

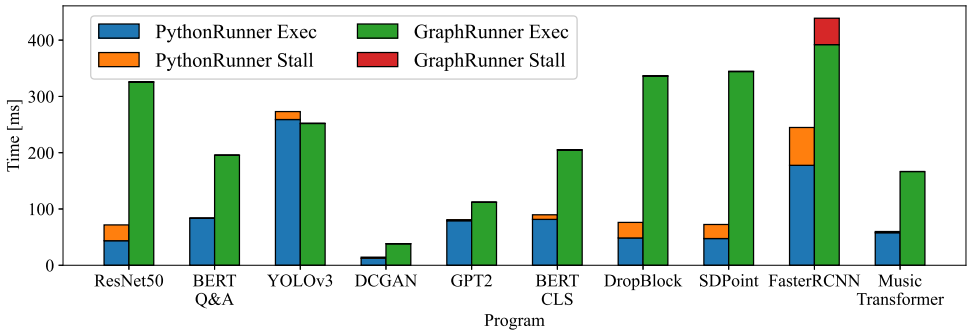


Figure 3.11 Performance breakdown within a single training step for both the PythonRunner and the GraphRunner.

execution phase because Terra's execution is mostly in this phase. The number of transitions between the two phases and the overhead analysis for the *tracing* phase are included in Appendix F. Figure 3.11 shows the performance breakdown of the two Runners in a single training step. 'PythonRunner Exec' denotes the Python interpreter's active running time, such as executing user code or validating the symbolic graph. Both 'PythonRunner Stall' and 'GraphRunner Stall' indicate the stall time in which the PythonRunner waits for the GraphRunner to fetch the materialized tensor or vice versa. Finally, we measure GPU's active time to run the CUDA kernels along with the overhead of TensorFlow's graph executor as 'GraphRunner Exec'. For all programs except for FasterRCNN, the GraphRunner is not stalled, implying that the GraphRunner fully exploits the optimized performance of the symbolic execution. In FasterRCNN, the stall of the GraphRunner occurs when the PythonRunner receives a materialized tensor from the GraphRunner and feeds it back to the GraphRunner. For YOLOv3, the PythonRunner's execution time is longer than that of the GraphRunner, which yields the slightly larger performance gap between Terra and AutoGraph in Figure 3.10.

Table 3.2 Comparison of the training speed-up between Terra and Terra with lazy evaluation. The results are relative speed-up to TensorFlow imperative execution as Figure 3.10.

Program	Terra	Terra LazyEval
ResNet50	x1.25	x1.13
BERT Q&A	x1.23	x0.94
DCGAN	x1.56	x1.34

Moreover, Figure 3.11 shows that the GraphRunner takes a longer time than the PythonRunner in most cases. The result implies the reason why the performance improvements of Terra are comparable to the performance improvements of AutoGraph. The execution of the PythonRunner is efficiently concealed by the execution of the GraphRunner with the co-execution. To demonstrate the effect of the co-execution, we serialize the execution of the PythonRunner and the GraphRunner then evaluate the performance for the simple programs among our benchmarks. Within the serialized execution, the GraphRunner does not start the execution along with the PythonRunner. Instead, it starts the execution when the PythonRunner requires tensor data through the *Output Fetching* operation. Eventually, the serialized execution is the same as the lazy evaluation that LazyTensor [102] does. Our results in Table 3.2 show that the lazy evaluation cannot fully achieve high performance of the symbolic execution. Even worse, it could become slower than the imperative execution when the execution time of the GraphRunner is not much longer than the execution time of the PythonRunner.

3.3.5 Tracing Phase Analysis

Table 3.3 shows the number of collected traces and the number of fallbacks from the *co-execution* of Terra. The results show that the symbolic graphs of all the

Table 3.3 Results of the number of collected traces and the number of fallbacks for each program.

Program	# Collected Traces	# Fallbacks
ResNet50 [106]	2	0
BERT-Q&A [22]	2	0
YOLOv3 [125]	2	0
DCGAN [105]	2	0
GPT2 [99]	2	0
BERT-CLS [53]	2	0
DropBlock [21]	3	0
SDPoint [50]	4	1
FasterRCNN [116]	2	0
MusicTransformer [44]	2	0

programs can be generated with at most four traces. This indicates that our approach–collecting multiple traces and incrementally generating the symbolic graph–is a plausible strategy. Furthermore, the number of collected traces for BERT-CLS, FasterRCNN, and MusicTransformer shows that Terra correctly executes the programs with only two traces, while AutoGraph cannot execute them at all.

3.4 Summary

We propose Terra, a novel approach to execute imperative Python DL programs. Terra performs imperative-symbolic co-execution, which addresses the problem of converting an imperative program to a symbolic graph completely. Terra generates a symbolic graph only from the DL operations of an imperative DL program. It then carries out the imperative execution, simultaneously executing the symbolic graph. Therefore, Terra achieves optimized performance while maintaining all Python features of the imperative program. Our evaluation shows that Terra can speed up all imperative DL programs, even for the

programs that AutoGraph cannot handle.

Chapter 4

Memory-Balanced Pipeline Parallelism for Training Large Language Models

4.1 Motivation

When training a LLM, increasing the degree of pipeline parallelism yields a *memory imbalance problem* between pipeline stages. Using the 1F1B pipeline schedule, as illustrated in Figure 2.2, the first stage has to store activations as many micro-batches as the pipeline parallelism degree during the warmup phase. For the other stages, the number of micro-batches in the warmup phase decreases linearly, so the last stage holds activations of a single micro-batch. Consequently, the imbalance of memory usage exists across the pipeline stages, and its magnitude amplifies with an escalation in the pipeline parallelism degree.

Figure 4.1 shows the memory usage of each pipeline stage when training a GPT-3 13B [9] model with 8-way pipeline parallelism using 8 NVIDIA A100 80 GiB GPUs. In 8-way pipeline parallelism, the first stage computes eight micro-

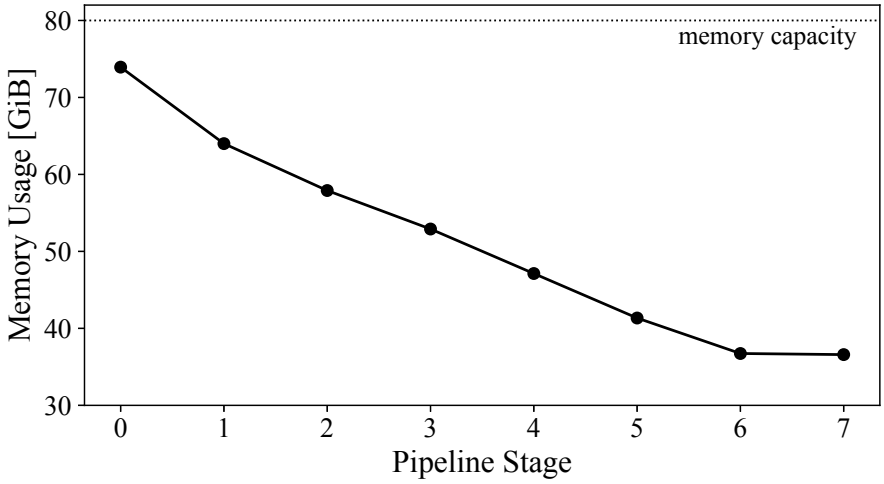


Figure 4.1 Memory imbalance among pipeline stages when training a GPT-3 13B model with 8-way pipeline parallelism. The micro-batch size is 1, and recomputation is not applied. Both the first and last stages are off the linear line because they require more memory due to the embedding and fully-connected layers, respectively. The memory difference between the first stage and the last stage is 37 GiB.

batches during the warmup phase. Therefore, the difference in the number of micro-batches between stage 0 and stage 7 is seven, causing a 37 GiB memory imbalance. In other words, stage 7 only utilizes up to half of the memory due to the imbalance. Even worse, attempting to accelerate training by utilizing the unused memory of the last pipeline stage (e.g., increasing the micro-batch size) might fail because the first stage no longer has enough memory.

Asymmetric partitioning of the pipeline stage (i.e., later stages contain more layers than earlier stages) might eliminate the imbalance, but it incurs inefficiency because the pipeline latency is minimized when each pipeline stage has the same computation time [68, 24, 60, 129]. Alternatively, Chimera [57] relieves the imbalance by replicating model parameters so that each GPU holds two pipeline stages. However, since replicating model parameters doubles the

memory usage, the overall memory usage increases for the same pipeline degree. To the best of our knowledge, BPipe is the first work that speeds up training LLMs by addressing the memory imbalance with smart activation transfer across GPUs.

4.2 Method

In this section, we describe an *activation balancing* method that solves the memory imbalance problem by transferring activations between pipeline stages. First, we formalize the memory imbalance and provide a high-level view of the activation balancing. Then, we define the balanced memory objective and demonstrate an activation transfer scheduling algorithm that achieves the objective. Finally, we describe how we assign pipeline stages to GPUs to minimize the transfer time.

4.2.1 Pipeline Memory Imbalance

On p-way pipeline parallelism with m micro-batches, the memory usage of the pipeline stage s can be expressed as $M(s) = W(s) + A(s)$, where $A(s)$ is the maximum amount of saved activations and $W(s)$ is the size of model parameters including optimizer states. To simplify, assume that a model has repetitive layers which account for almost all of the model parameters, such as Transformer models [20, 88, 9]. If all pipeline stages have an even number of layers, $W(s) = W_0$ is constant. Then, $A(s)$ that varies with the pipeline stage s determines the memory usage. If we define $\mu(s)$ as the maximum number of saved micro-batches on stage s, $A(s)$ becomes $A(s) = A_0\mu(s)$, where A_0 is the size of saved activations per micro-batch, which is also a constant value when each pipeline stage has the same number of layers. As a result, $M(s)$ can be written as the

following.

$$M(s) = W_0 + A_0\mu(s) \tag{4.1}$$

According to Figure 2.2, each pipeline stage in the 1F1B pipeline schedule possesses at most $\mu(s) = \min(p-s, m)$. In the practical case, $m \gg p$ is satisfied because such a case fully saturates all pipeline stages. Therefore, $\mu(s)$ of the 1F1B pipeline schedule satisfies the following relation.

$$\mu(s) = p - s \tag{4.2}$$

Substituting $\mu(s)$ in Eq. 4.1 with $p - s$, Eq. 4.1 denotes that the difference in the number of saved micro-batches is a cause of the imbalance of $M(s)$.

4.2.2 Activation Balancing

BPipe eliminates the memory imbalance by reducing the difference in the number of saved micro-batches for all pipeline stages. Eq. 4.1 and Eq. 4.2 show that the memory usage of the pipeline stage decreases linearly as the pipeline stage increases. Accordingly, we pair each stage s with stage $p-s-1$ to balance $\mu(s)$ and $\mu(p-s-1)$. The memory imbalance within each pair can be written as the following equation.

$$M(s) - M(p-s-1) = A_0(p-2s-1) \tag{4.3}$$

Eq. 4.3 implies that the pipeline stages with $s < \frac{p-1}{2}$ occupy more memory than the stages with $s > \frac{p-1}{2}$. We define the former stages as *evictor*s and the latter stages as *acceptor*s. Each pair then consists of an evictor and an acceptor, where the evictor evicts activations to its pair acceptor to balance $\mu(s)$ and $\mu(p-s-1)$.

Figure 4.2 illustrates the activation balancing within a 4-way 1F1B pipeline schedule with eight micro-batches. Now, assume we want to make $\mu(0)$ as 3.

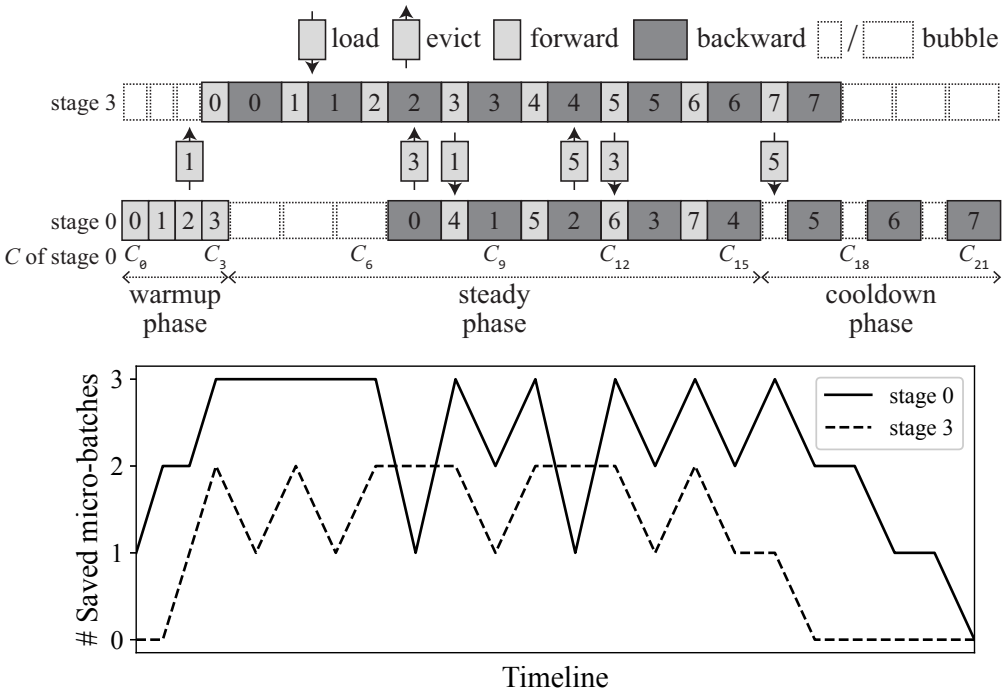


Figure 4.2 An illustration of the activation balancing and the corresponding changes in the number of saved micro-batches within a 4-way 1F1B pipeline schedule with eight micro-batches. Stage 0 and stage 3 are the pair evictor and acceptor, so stage 0 evicts activations to stage 3 and loads them before the backward computation. All transfers are running parallel to the forward or backward computation.

While stage 0 proceeds the forward computation of the third micro-batch, stage 0 evicts the saved activations to stage 3, a pair acceptor of stage 0. The number of saved micro-batches does not increase after the forward computation due to the eviction. BPipe always evicts the latest micro-batch among the saved to minimize the number of transfers. Therefore, the second micro-batch is evicted instead of the first micro-batch. The evicted activations should be loaded by stage 0 before processing the corresponding backward computation at the steady phase. Stage 0 evicts another micro-batch before loading because each forward computation and loading increase the number of saved micro-batches by 1, respectively. For example, evicting the micro-batches whose indices are 3 and 5 in Figure 4.2 represent it. At the cooldown phase, stage 0 loads the activations without the additional eviction because no forward computation is executed along with the loading. As a result, $\mu(0)$ becomes three, as shown in the graph of Figure 4.2.

4.2.3 Balanced Memory Objective

From Eq. 4.2, the sum of the maximum number of saved micro-batches for any evictor-acceptor pair is $\mu(s) + \mu(p - s - 1) = p + 1$. Including a buffer for the additional evictions at the steady phase, the value becomes $p + 2$. Now, the optimally balanced number of micro-batches μ_{opt} is derived as $\mu_{opt} = \lceil \frac{p+2}{2} \rceil$, and the optimal memory balance M_{opt} is $W_0 + A_0\mu_{opt}$.

An objective of the activation balancing is accomplishing $\mu(s) \leq \mu_{opt}$ for all pipeline stages throughout the training. An evictor with pipeline stage s achieves the objective by evicting at most $(p - s) - \mu_{opt} + 1$ micro-batches to its pair acceptor. Simultaneously, the pair acceptor saves at most $s + 1$ micro-batches following the pipeline schedule and $(p-s)-\mu_{opt}+1$ micro-batches from the evictor. $\mu(p - s - 1)$ then becomes $p + 2 - \mu_{opt}$, where the value is less or

equal to μ_{opt}. Therefore, both the evictor and acceptor achieve the objective.

The memory objective implies that BPipe does not trigger any transfer of activations for the evictors whose stages already satisfy the objective. In other words, the activation balancing operates when $p \geq 4$, and only the evictors whose pipeline stage satisfies $s \leq \lfloor \frac{p-4}{2} \rfloor$. For example, the innermost pair of pipeline stages, which is stage 1 and stage 2 in Figure 4.2, does not transfer activations because they already satisfy the objective.

4.2.4 Transfer Schedule

To achieve the memory objective, we propose a transfer scheduling algorithm. The algorithm gets a pipeline parallelism degree p, a pipeline stage s of an evictor, the number of micro-batches m, and a computation schedule C of 1F1B pipeline as inputs and returns a transfer schedule T for stage s. A computation schedule is an array of *computation decisions*. Each computation decision C_i comprises the type of computation made and the index of the micro-batch being processed. For example, C of pipeline stage 0 in Figure 4.2 has 22 computation decisions, including pipeline bubbles. C_0 has 0 as a micro-batch index with a *FORWARD* computation type, C_9 has 1 as a micro-batch index with a *BACKWARD* computation type, and C_{18} has an empty micro-batch index because the computation type of C_{18} is a *BUBBLE*.

Algorithm 2 describes the details of scheduling for the given inputs. The algorithm traverses the computation schedule, deciding when to evict or load. Within the warmup phase, *n_evict* micro-batches are evicted to make $\mu(s)$ equal to μ_{opt} (lines 9-13). When the algorithm encounters a *BACKWARD* type computation decision at i and finds that the micro-batch required to compute the backward (i.e., $C_i.idx$) has been evicted, it loads the micro-batch at $i-1$ (lines 14-18). However, this load would make $\mu(s)$ exceed μ_{opt} if C_i belongs to

Algorithm 2: Transfer Scheduling Algorithm

```
Input: p, s, m, 1F1B computation schedule C
Output: transfer schedule T
μ_opt ← ⌈(p+2)/2⌉
n_evict ← min(p − s, m) − μ_opt
evicted ← ∅
T_i ← None ∀0 ≤ i < |C|
for i = 0 to |C| − 1 do
  if C_i.type = FORWARD then
    if μ_opt − 1 ≤ C_i.idx < μ_opt − 1 + n_evict then
      /* warmup phase */
      T_i ← Evict(C_{i−1}.idx)
      evicted ← evicted ∪ {C_{i−1}.idx}
    end if
  else if C_i.type = BACKWARD then
    /* steady or cooldown phase */
    if C_i.idx ∈ evicted then
      T_{i−1} ← Load(C_i.idx)
      evicted ← evicted \ {C_i.idx}
      if C_{i−1}.type = FORWARD then
        /* steady phase */
        /* evict to keep μ(s) ≤ μ_opt */
        T_{i−2} ← Evict(C_{i−3}.idx)
        evicted ← evicted ∪ {C_{i−3}.idx}
      end if
    end if
  end if
end for
```

the steady phase. In other words, when C_{i-1} is a *FORWARD* type, both C_{i-1} and the load at $i-1$ increase $\mu(s)$ so that $\mu(s)$ exceeds μ_{opt}. The algorithm prevents it by evicting an additional micro-batch that will be needed at the furthest future at $i-2$, whose index is $C_{i-3}.idx$ (lines 19-23). On the other hand, if C_i belongs to the cooldown phase, C_{i-1} is the *BUBBLE* type which does not increase $\mu(s)$. Thus, $\mu(s)$ does not exceed μ_{opt}, and the algorithm does not evict an additional micro-batch.

The generated transfer schedule T contains an array of *transfer decisions* that interleaved with the 1F1B computation schedule. While processing the 1F1B schedule, BPipe simultaneously processes the transfer schedule. BPipe evicts the saved activations of the j-th micro-batch if T_i is $Evict(j)$ and loads them if T_i is $Load(j)$. When T_i is $None$, a default decision in line 6, BPipe does not process any transfer.

Following the computations with the corresponding transfer decisions, BPipe achieves the memory objective for all pipeline stages with the minimum number of transfers. The number of transfers is the summation of the number of *Evict*s and *Load*s. We only consider the number of *Evict*s because each eviction requires exactly one corresponding *Load* before processing the backward computation. Subsequently, we can interpret the micro-batch eviction as analogous to cache eviction. In other words, an evictor manages a cache storage whose capacity is μ_{opt}. It evicts an item (i.e., forward micro-batch) to the memory space of its pair acceptor when the cache is full. Deciding which item to evict is then determined based on a cache eviction policy. Unlike common caching scenarios, we know exactly when each item is required in the future. Therefore, as evicting the item that will be needed in the furthest future is optimal [7], our scheduling algorithm minimizes the number of transfers.

Although the algorithm assumes a 1F1B computation schedule as an input,

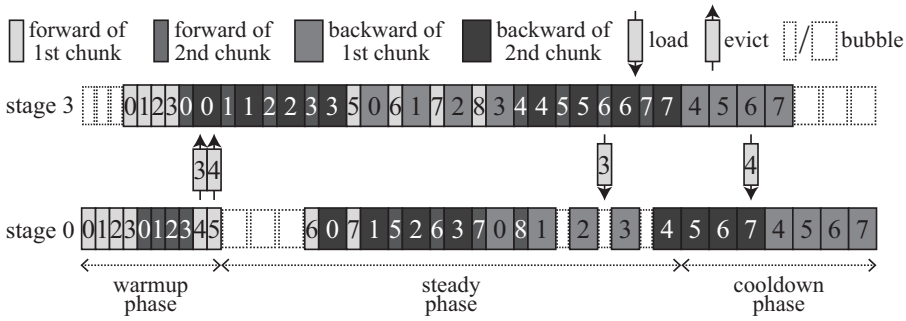
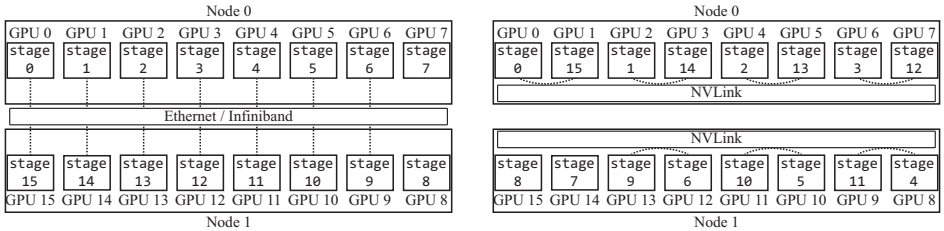


Figure 4.3 An illustration of the activation balancing within a 4-way interleaved 1F1B pipeline schedule with eight micro-batches and two model chunks for each pipeline stage.

we can extend it to other variants of 1F1B [69, 70, 121, 5, 133] because all of them have a memory imbalance. As an representative example, we descibe how we can apply the algorithm to the interleaved 1F1B pipeline schedule [70]. In the interleaved schedule, each pipeline stage carries out the computations of model chunks, which are non-contiguous subsets of layers [70]. Assume a pipeline stage s of p-way pipeline parallelism. When each pipeline stage has v model chunks, $p(v-1)+2(p-s-1)+1$ micro-batches are calculated before the first backward computation. Then, μ_{opt} is derived as the following.

$$\mu_{opt} = \lceil \frac{p(v-1)+2(p-s-1)+1+p(v-1)+2(p-(p-s-1)-1)+1+1}{2} \rceil$$
$$= pv+1$$

Now, the transfer scheduling of the interleaved 1F1B pipeline schedule is similar to that of the vanilla 1F1B schedule. Figure 4.3 illustrates how the activation balancing operates to the interleaved 1F1B pipeline schedule. Within the warmup phase, an evictor evicts the $p-2(s+1)$ micro-batches to its pair acceptor. In the steady or cooldown phase, it loads the evicted micro-batches to perform the backward computations.



(a) A standard assignment (b) A pair-adjacent assignment.

Figure 4.4 An illustration of both standard assignment and pair-adjacent assignment for 16-way pipeline parallelism on 2 nodes, each with 8 GPUs. The dotted lines represent the communication between evictor-acceptor pairs. The standard assignment makes each pair communicate over the slow inter-node link. In contrast, the pair-adjacent assignment let them transfer their activations over the fast intra-node link.

4.2.5 Pair-Adjacent Assignment

As BPipe processes a transfer schedule simultaneously with a 1F1B computation schedule, each transfer should take less time than *FORWARD* or *BACKWARD* not to affect the training time. To minimize the transfer time, we propose a *pair-adjacent assignment* that locates each evictor-acceptor pair in the same node in a cluster. Within the same node, each pair can exploit a high-bandwidth intra-node communication link like NVLink rather than using a relatively slower inter-node communication link like Ethernet or InfiniBand.

Figure 4.4 illustrates how pipeline stages are assigned to GPUs using 16-way pipeline parallelism on a cluster with two nodes, each of which has 8 GPUs. A standard assignment assigns pipeline stages to GPUs in order, as Figure 4.4(a) shows. However, it assigns all pipeline pairs to different nodes. Instead, BPipe assigns pair stages to adjacent GPUs, as shown in Figure 4.4(b). Each pair then resides in the same node and utilizes fast intra-node communication. Although the pair-adjacent assignment entails additional inter-node communication of dependent data between consecutive pipeline stages, it is beneficial because

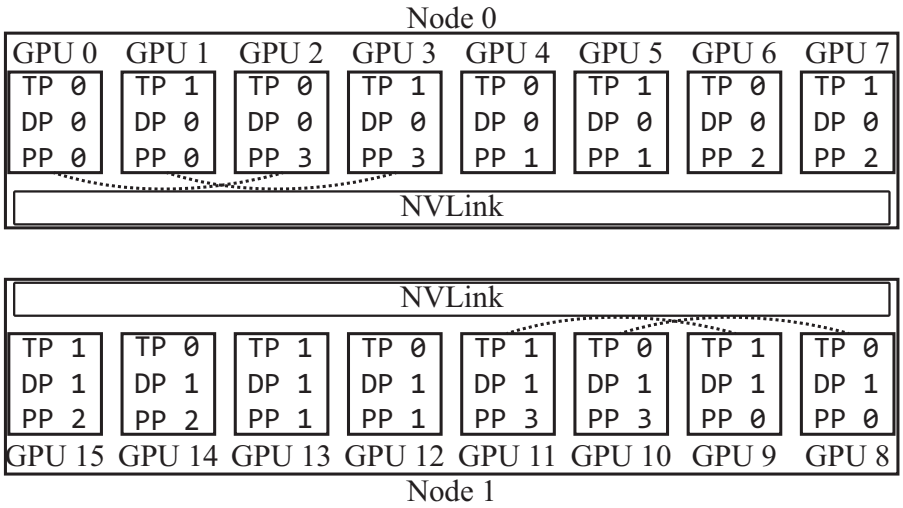


Figure 4.5 An illustration of how the pair-adjacent assignment assigns pipeline stages to GPUs when 4-way pipeline parallelism with 2-way tensor parallelism and 2-way data parallelism on two nodes, each with 8 GPUs. 'TP' and 'DP' denote the rank of tensor parallelism and data parallelism. 'PP' represents the pipeline stage.

the bytes transferred between pairs are much bigger than the bytes exchanged between consecutive stages.

When pipeline parallelism is used with either tensor or data parallelism, the pair-adjacent assignment assigns pipeline stages considering other parallelism methods. Figure 4.5 illustrates how the pair-adjacent assignment operates with data and tensor parallelisms. BPipe prioritizes pipeline parallelism over data parallelism because the gradient synchronization of data parallelism is performed only once for each training step. On the other hand, BPipe prioritizes tensor parallelism over pipeline parallelism to ensure that the frequent collective communication of tensor parallelism always takes place within a node. Hence, when the tensor parallelism degree is equal to the number of GPUs in a single node, BPipe transfers activations across the node boundary. Nevertheless,

Table 4.1 Model configurations for evaluation. L denotes the number of layers, D denotes hidden dimension size, and H denotes the number of attention heads. Finally, G and B are the numbers of GPUs used to execute the model and batch size, respectively.

Model	L	D	H	G	B
GPT-3 13B	40	5120	40	8	32
GPT-3 96B	80	9984	104	32	128
GPT-3 134B	84	11520	120	48	192

the activation balancing is feasible across the node boundary in our evaluation setup.

4.3 Evaluation

To evaluate BPipe, we ask the following questions.

- Does BPipe facilitate faster training of large language models? (§ 4.3.2)
- Does BPipe flatten the memory usage of each pipeline stage? (§ 4.3.3)
- Does BPipe efficiently evict and load activations without performance degradation? (§ 4.3.4)

4.3.1 Implementation and Environment Setup

We have implemented BPipe on Megatron-LM v3 [48]. We utilize separate CUDA streams for evicting and loading activations so that activations can be transferred concurrently with either forward or backward computation. In addition, we manually manage CUDA memory for activations that are needed for backward computation and reuse pre-allocated memory after the eviction to avoid unexpected memory fragmentation.

Our evaluations are conducted on a cluster of six HPE Apollo 6500 Gen10 Plus nodes, each of which is equipped with 8 NVIDIA 80 GiB A100 GPUs

connected over NVLink and 4 Mellanox 200 Gbps HDR InfiniBand HCAs for communication. All experiments are executed on the NVIDIA PyTorch NGC 22.09 container. We evaluate GPT-3 [9] throughout the experiments, one of the most representative LLMs. We use three different model configurations, as shown in Table 4.1, in which the largest model has 134 billion parameters in total. Sequence length and vocabulary size are 2,048 and 51,200 for all models, respectively, and we use mixed precision training [64]. Although we evaluate only GPT-3 models, BPipe is applicable to any model as long as pipeline parallelism is used to train the model.

4.3.2 Training Performance

To evaluate whether BPipe can accelerate training, we find the fastest configuration for training GPT-3 96B and GPT-3 134B models, with and without BPipe. Our baseline is Megatron-LM v3 [48], a state-of-the-art LLM training framework. We perform a grid search to find the best configuration as follows. In addition to the notations in Table 4.1, we define tensor parallelism degree as t, pipeline parallelism degree as p, data parallelism degree as d, and micro-batch size as mb. Then, the following constraints exist.

- $H \mod t = 0$
  - The number of attention heads should be divisible by the tensor parallelism degree because tensor parallelism for Transformer layers splits attention heads.
- $8 \mod t = 0$ (8 is the number of GPUs in a single node)
  - Tensor parallelism is practical when used within a node boundary due to its costly synchronization.

Table 4.2 Training configurations of GPT-3 96B and GPT-3 134B models. 'tensor' and 'pipeline' represent the tensor and pipeline parallelism degrees, respectively. The remaining GPUs are used for data parallelism, and we use ZeRO stage-1 data parallelism [90] that splits optimizer states. Moreover, tensor parallelism includes partitioning layer normalization and dropout [48]. 'mb' denotes the micro-batch size, and each value corresponds to a different training configuration.

Model	Model Parallelism			mb	Recompute scope
	ID	tensor	pipeline		
GPT-3 96B	(1)	1	16	1	layer
	(2)	2	8	1,2	layer
	(3)	4	4	1	attention
				2	layer
	(4)	8	2	1	attention
				2,4	layer
	(5)	2	16	1	attention
				2,4	layer
	(6)	4	8	1,2	attention
				4,8	layer
	(7)	8	4	1,2	attention
				4,8	layer
GPT-3 134B	(1)	2	12	1,2	layer
	(2)	4	6	1	attention
				2,4	layer
	(3)	8	3	1	attention
				2,4	layer
	(4)	4	12	1,2	attention
				4	layer
	(5)	8	6	1,2	attention
				4,8	layer

- $L \mod p = 0$
  - The number of layers should be divisible by the pipeline parallelism degree when pipeline parallelism evenly divides the layers.
- $B \mod (mb \times d) = 0$
  - The total batch size should be divisible by the micro-batch size times data parallelism degree. The number of micro-batches is then computed as $B/(b \times d)$.
- $t \times d \times p = G$
  - Multiplication of the degrees of tensor, data, and pipeline parallelism should be equal to the total number of GPUs.

Under the constraints, we enumerate all possible tuples of (t, p, d, mb). We evaluate them with Megatron-LM for each recomputation scope in the order of none, attention, and layer, where the none scope does not recompute any activation, the attention scope recomputes only the self-attention of the Transformer layer, which is known as the selective recomputation [48], and the layer scope recomputes the entire Transformer layer. We apply early stopping when succeeding to execute with fewer recomputations because carrying out more recomputations for the same (t, p, d, mb) is inefficient. Then, each (t, p, d, mb) with a recomputation scope composes a single training configuration. For BPipe, we evaluate the configurations that Megatron-LM cannot execute since the activation balancing does not accelerate training. If BPipe fails due to the out-of-memory error, we exclude those configurations. As a result, Table 4.2 lists all feasible training configurations.

We use model FLOPS utilization (MFU) as an evaluation metric, a ratio of the observed throughput to the hardware maximum throughput [48]. Table 4.3

Table 4.3 MFU numbers of GPT-3 96B and GPT-3 134B models. The numbers are the values of Megatron-LM, except the values that are annotated with **BPipe**. For those configurations, Megatron-LM fails to run due to the out-of-memory error.

Model	Model Parallelism ID	tensor	pipeline	mb	Recompute Scope	MFU [%]
GPT-3 96B	(1)	1	16	1	layer	38.08
	(2)	2	8	1	layer	40.46
				2	layer	30.29
	(3)	4	4	1	attention	37.59
				2	layer	40.51
	(4)	8	2	1	attention	35.47
				2	layer	30.07
				4	layer	38.08
	(5)	2	16	1	attention	**48.78 (BPipe)**
				2	layer	36.78
				4	layer	29.79
	(6)	4	8	1	attention	37.09
				2	attention	**50.64 (BPipe)**
				4	layer	36.23
				8	layer	25.74
	(7)	8	4	1	attention	36.18
				2	attention	36.56
				4	layer	38.04
				8	layer	24.31
GPT-3 134B	(1)	2	12	1	layer	39.98
				2	layer	38.64
	(2)	4	6	1	attention	39.90
				2	layer	41.40
				4	layer	30.06
	(3)	8	3	1	attention	37.77
				2	layer	31.76
				4	layer	38.81
	(4)	4	12	1	attention	39.19
				2	attention	**52.06 (BPipe)**
				4	layer	37.22
	(5)	8	6	1	attention	38.45
				2	attention	38.72
				4	layer	38.72
				8	layer	24.01

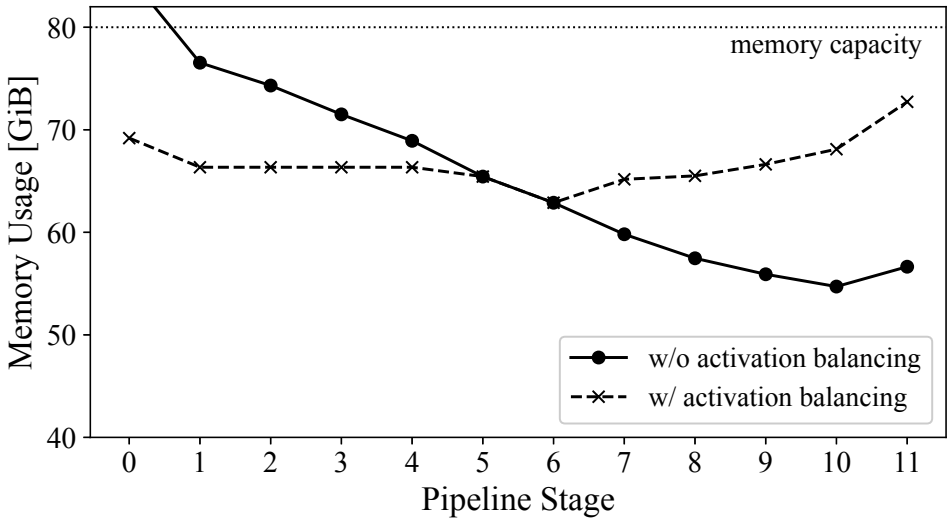


Figure 4.6 Memory usage of each pipeline stage with the activation balancing compared to without the activation balancing when training a GPT-3 134B model with configuration (4)-mb2 of Table 4.2. To estimate the memory usage without the activation balancing, we allow the activation balancing only for stage 0 and stage 11 since stage 0 has insufficient memory to execute.

shows that BPipe can execute the configuration that achieves 52.06% MFU, though Megatron-LM cannot execute it due to the memory imbalance problem. Consequently, BPipe accelerates training the models by 1.25x compared to the fastest training configuration that Megatron-LM can execute and 2.17x than the most inefficient configuration of Megatron-LM.

4.3.3 Memory Balancing

Figure 4.6 presents the change in memory usage when using BPipe. Without the activation balancing, the maximum memory usage difference between the pipeline stages is larger than 25 GiB. The model cannot be executed because the first stage runs out of memory. However, the difference sharply reduces to 10 GiB with BPipe because BPipe flattens the memory usage of each evictor-

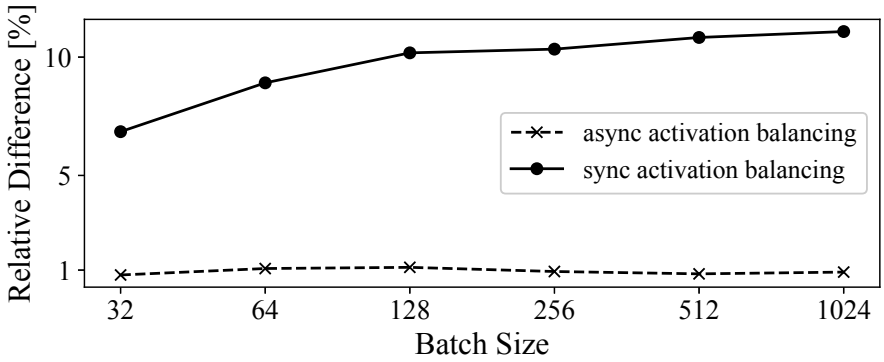


Figure 4.7 The relative difference in iteration time with various batch sizes when training a GPT-3 13B with 8-way pipeline parallelism. No recomputation is applied and the number of micro-batches is equal to the batch size.

acceptor pair. As a result, BPipe facilitates faster training with more efficient resource utilization, as Table 4.3 shows.

If the model size grows, the pipeline parallelism degree should increase because increasing the tensor parallelism degree is bounded up to the number of GPUs in a single node. Then, the memory imbalance would also increase following the pipeline parallelism degree. For example, Megatron-LM v2 [70] and v3 [48] use 64-way pipeline parallelism for scaling up GPT-3 model to 1 trillion parameters. It implies that BPipe can dramatically flatten the memory imbalance, paving the way for more efficient training of LLMs.

4.3.4 Performance Analysis

We inspect the efficiency of BPipe by comparing the iteration time before and after applying the activation balancing. Figure 4.7 shows the additional time cost of the activation balancing, varying the batch size from 32 to 1,024 when training a GPT-3 13B model with 8-way pipeline parallelism. The cost occupies 1.1% even when the number of micro-batches is sufficiently large.

Table 4.4 Time breakdown of transfer, forward, and backward. All times are the elapsed time for processing a single micro-batch. For GPT-3 13B, 8-way pipeline parallelism is used with the attention recomputation scope and the micro-batch size is 1. For GPT-3 96B and GPT-3 134B, we use configuration (6)-mb2 and (4)-mb2 of Table 4.2, respectively.

Model	transfer	forward	backward
GPT-3 13B	7.63 ms	36.32 ms	83.57 ms
GPT-3 96B	15.63 ms	143.37 ms	287.90 ms
GPT-3 134B	12.06 ms	126.87 ms	256.30 ms

Table 4.5 Time breakdown by varying the recomputation scope of a GPT-3 13B model with 8-way pipeline parallelism.

Recompute scope	transfer	forward	backward
none	22.48 ms	36.32 ms	74.12 ms
attention	7.63 ms	36.32 ms	83.57 ms
layer	0.58 ms	36.32 ms	110.33 ms

BPipe achieves a low time cost by virtue of the asynchronous activation transfer. If we transfer activations synchronously, the time overhead shoots up to 11%, as Figure 4.7 shows. On the contrary, asynchronous transfer overlaps with the computation because the computation time takes far longer than the transfer time. The transfer overlaps even if the model size grows because the computation time increases more steeply than the transfer time, as Table 4.4 shows. Furthermore, the size of activations to transfer increases with fewer recomputations. However, the transfer time does not exceed the forward time even for the no recomputation, as Table 4.5 shows.

4.3.5 Communication Bandwidth Requirement for Transfer

Since each transfer of the activation balancing should finish earlier than the forward computation of a single micro-batch, we can calculate the required

Table 4.6 Definition of the variables.

Variable	Definition
L	number of layers
H	number of attention heads
D	hidden dimension size
l	sequence length
p	pipeline parallelism degree
t	tensor parallelism degree
b	micro-batch size
f	forward computation time of a single micro-batch [sec]

network bandwidth of a single GPU with the forward computation time. Using the variables in Table 4.6, we can derive the bandwidth requirements as the following. According to Megatron-LM v3 [48], the bytes of activation memory per each forward micro-batch are:

$$\text{No recomputation: } \frac{Llbh}{pt}(34 + \frac{5Hl}{h}) \text{ bytes} \tag{4.4}$$

$$\text{Attention recomputation scope: } 34\frac{Llbh}{pt} \text{ bytes} \tag{4.5}$$

$$\text{Layer recomputation scope: } 2\frac{Llbh}{p} \text{ bytes} \tag{4.6}$$

Dividing Eq (4.4) to (4.6) by f, bandwidth requirements are derived as the following equations.

$$\text{No recomputation: } \frac{1}{10^9}\frac{Llbh}{ptf}(34 + \frac{5Hl}{h}) \text{ GB/s} \tag{4.7}$$

$$\text{Attention recomputation scope: } \frac{34}{10^9}\frac{Llbh}{ptf} \text{ GB/s} \tag{4.8}$$

$$\text{Layer recomputation scope: } \frac{2}{10^9}\frac{Llbh}{pf} \text{ GB/s} \tag{4.9}$$

If a single node has 8 GPUs, two GPUs within the node where an evictor-acceptor pair resides should have larger bandwidth than Eq (4.7) to (4.9) when the tensor parallelism degree is 1, 2, or 4. For the tensor parallelism degree of 8, inter-node bandwidth divided by 8 should be larger than the derived bandwidths.

In our evaluation, the fastest configuration of BPipe for the GPT-3 96B model uses 4-way tensor parallelism, 8-way pipeline parallelism, attention recomputation scope, and the micro-batch size of 2. Then the total bytes to transfer are 3.48 GB. Since each forward computation takes 143.37 ms in Table 4.4, the required bandwidth is 24.25 GB/s. As the forward computation time does not change by the recomputation scope, we can derive that NVLink is also sufficient for no recomputation in which the required bandwidth is 100.31 GB/s.

When the tensor parallelism degree is equal to the number of GPUs in a single node, BPipe transfers activations across the node boundary. However, the inter-node transfer is feasible if (1) a cluster has a fast inter-node network such as InfiniBand and (2) recomputation is used for training. Moreover, we alleviate the bandwidth requirements with the following optimization. In general, backward computation takes twice as long as forward computation. Therefore, we allow the consecutive *Evict-Load* to be performed over the timespan of consecutive *BACKWARD-FORWARD* computation within the steady phase. For example, in Figure 4.2, evicting and loading the micro-batches whose indices are 3 and 1 can be completed within the sum of the time required for backward and forward computations. The bandwidth requirements are then relieved as the following equations.

$$\text{No recomputation: } \frac{1}{10^9}\frac{2Llbh}{3ptf}(34 + \frac{5Hl}{h}) \text{ GB/s} \tag{4.10}$$

Table 4.7 MFU numbers of the GPT-3 96B model, when the tensor parallelism degree is 8.

Model	Model Parallelism ID	tensor	pipeline	mb	Recompute scope	Megatron MFU [%]	BPipe MFU [%]
GPT-3 96B	(7)	8	4	1	attention	36.18	35.80
				2	attention	36.56	36.52
				4	layer	38.04	38.00
				8	layer	24.31	27.47

$$\text{Attention recomputation scope: } \frac{68}{3 \cdot 10^9} \frac{Llbh}{ptf} \text{ GB/s} \tag{4.11}$$

$$\text{Layer recomputation scope: } \frac{4}{3 \cdot 10^9} \frac{Llbh}{pf} \text{ GB/s} \tag{4.12}$$

Table 4.7 presents the MFU values when the tensor parallelism degree is 8 for the GPT-3 96B model. The results show that BPipe is feasible even when the tensor parallelism degree is 8. Furthermore, when the micro-batch size is 8, the MFU value of BPipe is higher than the MFU value of Megatron-LM. For such a large micro-batch size, we observed that PyTorch periodically vacates the cache allocator due to high memory pressure, resulting in performance degradation. BPipe partially avoids such inefficiency with the activation balancing.

4.4 Summary

We propose BPipe, a memory-balanced pipeline parallelism method for training LLMs. With the activation balancing that transfers intermediate activations between pipeline stages, BPipe resolves the memory imbalance problem of pipeline parallelism. While training, all pipeline stages utilize a comparable amount of memory by storing a balanced amount of activations in the unit of

micro-batch. Our evaluation shows that BPipe speeds up training large-scale GPT-3 models by executing faster training configurations that are not feasible without BPipe.

Chapter 5

Related Work

Symbolic representation and DL compiler To broaden the possibilities of optimizations and to support various DL accelerators transparently, intermediate representations and compilers for DL have been proposed [1, 13, 19, 54, 110, 89, 113, 17, 128, 74, 93, 26, 42, 130]. All of them define a fixed set of operations that could be lowered to executable bytecodes for various hardware. Furthermore, they introduce more fine-grained primitives for DL operations and contribute to more efficient kernel implementations [13, 110, 54]. For a computation graph that a set of intermediate representations define, DL compilers optimizer either intra-operation level or inter-operations level. Since Terra creates a symbolic graph from an imperative DL program, Terra is instrumental in utilizing such representations and applying optimizations to an arbitrary DL program.

Kernel optimization DL models are composed of typical operations such as matrix multiplication and convolution. Thus, it is worthwhile to optimize spe-

cific kernels to obtain higher training performance. For example, cuBLAS [72] and cuDNN [15] offer de facto kernels when using GPUs to train. Similar to domain-specific language [13, 110, 54] that offer more fine-grained primitives for the operations, CUTLASS [73] provides template abstractions that can be used as building blocks within manual kernels. Recently, optimized implementations [98, 115, 86, 18] of the self-attention in the Transformer layer [114] exist after it became the essential building block of DL models.

Data parallelism Data parallelism replicates [59] or shares [56] model parameters and optimizer states across accelerators. An input batch is divided into multiple mini-batches for each training step, so each device conducts forward-backward computation with a different mini-batch. After each device finishes a single training step, all devices carry out an all-reduce collective communication to synchronize gradients and then update the model parameters [59]. Thereby, all devices always have the same parameters throughout the training. Whenever a model fits into a single device, increasing the data parallelism degree is the best way to scale up the training.

Beyond replicating model parameters, ZeRO [90] and Fully Sharded Data Parallelism (FSDP) [127] reduce the memory pressure by splitting optimizer states. They propose more aggressive splits, including gradients and model parameters, with more frequent collective communication.

Pipeline parallelism To further minimize the pipeline bubble, various pipeline schedules [69, 70, 57, 121, 5, 133] are proposed. Such schedules stem from the 1F1B schedule but sacrifice memory usage or even model correctness. Alternatively, it is possible to reduce the bubble in certain training scenarios. For example, token-level scheduling [60] is proposed for an auto-regressive language

model [9]. However, it is effective when the batch size is much smaller than the pipeline parallelism degree, which is unlikely in practical LLM training scenarios. TSPipe [61] reduces the bubble in the teacher-student knowledge distillation scenario by saturating forward and backward computations of both teacher and student models.

Tensor parallelism Mesh-TensorFlow [97] introduces an abstraction that can express arbitrary operation partitioning. It also presents that data parallelism is one of the cases of tensor parallelism that splits tensors across the batch dimension. Moreover, some model-specific tensor parallel strategies exist that efficiently reduce memory pressure with moderate synchronization costs. For example, Megatron-LM [98] proposes an efficient partitioning strategy of two consecutive matrix multiplications in the Transformer layer. Additionally, partitioning along the sequence dimension of the Transformer layer is also feasible [58, 48].

Other memory saving approaches Throughout forward computation, the memory pressure of a device increases since the activations should be reserved for the corresponding backward computation. This memory pressure could be reduced by releasing the activations after the forward computation and recomputing them during the backward computation [28, 47]. Since recomputation incurs an overhead equal to the forward computation time, which activations to discard and reserve should be well-decided. For a linear chain of computation, Chen et al. [14] proposed an algorithm that can reduce memory consumption to sub-linear cost. Checkmate [34] solves a mixed integer linear program to find an optimal recomputation strategy for an arbitrary model structure. When models are constructed with repetitive layers, recomputing activations layer by layer

is widely adopted in practice. In addition to the layerwise strategy, recent research [48] has shown that the memory pressure of Transformer models could be efficiently reduced by recomputing only their attentions.

On the other side, various systems have been proposed that utilize CPU memory. In general, a CPU has orders of magnitude larger and cheaper memory than a GPU. Hence, such systems exploit CPU memory as a swap storage of limited GPU memory [95, 117, 81, 79, 32, 94, 91]. However, utilizing CPU memory is not scalable because communication between CPU and GPU is slow due to the limited communication bandwidth of PCIe. Therefore, CPU offloading is worthy of consideration when the number of GPUs that could be utilized is very small compared to the size of the model to train.

Communication optimization In distributed training, the time proportion of the communication is not negligible with respect to the computations. Therefore, we can perform more efficient training by optimizing communication. For data parallelism, ByteScheduler [80] proposes a generic communication scheduler. It efficiently overlaps computation and communication based on Bayesian optimization.

Communication pattern becomes more complex with various parallelism methods. To deal with this, CoCoNet [35] provides an abstraction and a compiler stack that jointly optimize computation and communication. For Transformer models [20, 88, 76], Megatron-LM [70] reduces the bytes for point-to-point communication when both tensor and pipeline parallelism methods are used. Moreover, it overlaps the synchronization of tensor parallelism with backward computation to minimize the synchronization overhead of tensor parallelism.

Automatic search for the optimal training configuration Beyond the DL compilers [13, 39, 128] that only consider computations of a given DL model, various systems [118, 40, 55, 92, 103, 120, 8, 43, 37, 129, 112] exist that automatically search for the optimal execution plan for a pair of a DL model and a cluster environment. Such systems target large-scale model training including LLMs because writing a distributed DL program is challenging, and manually exploring all possible distributed configurations is painful for users. Those systems try to extend the abstraction of DL operations to cover the primitives of parallelism methods. Therefore, they configure a unified search space and explore it to find the best result. As BPipe rewrites the memory usage of pipeline parallelism, BPipe can expand the search space and help find better configurations.

Chapter 6

Future Work & Conclusion

6.1 Future Work

As the size of DL models continuously grows, DL frameworks need to be aware of the programming interface for distributed training. Although modern DL frameworks such as HuggingFace [119] and PyTorch Lightning [23] provide a modularized programming interface, users should struggle to apply various parallelism methods without expert knowledge. Especially for pipeline parallelism, it is hard to express the semantics with a simple abstraction because each pipeline stage executes a different program with different data. Therefore, distributed-friendly abstractions are being redesigned [123, 129, 112, 6]. Based on the knowledge that we have learned from Terra and BPipe, we are going to research a distributed DL framework that provides an expressive abstraction and facilitates an efficient training of large-scale deep learning models.

Furthermore, applying pipeline parallelism to DL models that have complicated dataflow is not trivial. For example, unlike the encoder-only model [20]

or decoder-only model [9], the encoder-decoder model [88] needs to send the output of the encoder to all layers of the decoder. Moreover, multi-modal models [87, 96] have heterogeneous model components. Since each of them has a different structure, naive partitioning could incur inefficient training. Thus, we will investigate the applicability of pipeline parallelism for various DL models and then study the best policy.

6.2 Conclusion

In this dissertation, we propose two systems for efficient training of DL models. First, Terra is a novel execution model of an imperative DL program. Terra preserves the convenience of the imperative execution while facilitating the optimizations of the symbolic execution with the imperative-symbolic co-execution. Therefore, Terra resolves the limited program coverage and the unsoundness of the existing systems. Our evaluation shows that Terra can accelerate the training of DL programs, even for the programs that the existing systems fail to execute. In addition, we propose BPipe, a novel pipeline parallelism method for training LLMs. The current pipeline parallelism entails the memory imbalance problem, in which all pipeline stages reserve asymmetric amounts of memory. It disturbs efficient training because some pipeline stages could run out of memory. BPipe deals with the memory imbalance problem by transferring intermediate activation, making all pipeline stages utilize comparable amounts of memory. As a result, the evaluation results of BPipe demonstrate that BPipe reserves room for more efficient training with a balanced memory load.

Bibliography

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: a system for large-scale machine learning. In *Osdi*, volume 16, pages 265–283. Savannah, GA, USA, 2016.

[2] A. Agrawal, A. Modi, A. Passos, A. Lavoie, A. Agarwal, A. Shankar, I. Ganichev, J. Levenberg, M. Hong, R. Monga, et al. Tensorflow eager: A multi-stage, python-embedded dsl for machine learning. *Proceedings of Machine Learning and Systems*, 1:178–189, 2019.

[3] B. H. Ahn, J. Lee, J. M. Lin, H. Cheng, J. Hou, and H. Esmaeilzadeh. Ordering chaos: Memory-aware scheduling of irregularly wired neural networks for edge devices. In *MLSys*. mlsys.org, 2020.

[4] R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, A. Belopolsky, et al. Theano: A python framework for fast computation of mathematical expressions. *arXiv e-prints*, pages arXiv–1605, 2016.

[5] S. Athlur, N. Saran, M. Sivathanu, R. Ramjee, and N. Kwatra. Varuna: scalable, low-cost training of massive deep learning models. In *Proceed-*

ings of the Seventeenth European Conference on Computer Systems, pages 472–487, 2022.

[6] P. Barham, A. Chowdhery, J. Dean, S. Ghemawat, S. Hand, D. Hurt, M. Isard, H. Lim, R. Pang, S. Roy, B. Saeta, P. Schuh, R. Sepassi, L. E. Shafey, C. A. Thekkath, and Y. Wu. Pathways: Asynchronous distributed dataflow for ML. In *MLSys*. mlsys.org, 2022.

[7] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

[8] Z. Bian, H. Liu, B. Wang, H. Huang, Y. Li, C. Wang, F. Cui, and Y. You. Colossal-ai: A unified deep learning system for large-scale parallel training. *arXiv preprint arXiv:2110.14883*, 2021.

[9] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[10] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, et al. Api design for machine learning software: experiences from the scikit-learn project. *arXiv preprint arXiv:1309.0238*, 2013.

[11] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[12] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.

[13] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Q. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. TVM: an automated end-to-end optimizing compiler for deep learning. In A. C. Arpaci-Dusseau and G. Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 578–594. USENIX Association, 2018.

[14] T. Chen, B. Xu, C. Zhang, and C. Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[15] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014.

[16] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. Rao, P. Barnes, Y. Tay, N. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. Garcia, V. Misra, K. Robinson, L. Fedus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Diaz, O. Firat, M. Catasta, J. Wei,

K. Meier-Hellstern, D. Eck, J. Dean, S. Petrov, and N. Fiedel. Palm: Scaling language modeling with pathways. *CoRR*, abs/2204.02311, 2022.

[17] S. Cyphers, A. K. Bansal, A. Bhiwandiwalla, J. Bobba, M. Brookhart, A. Chakraborty, W. Constable, C. Convey, L. Cook, O. Kanawi, R. Kimball, J. Knight, N. Korovaiko, V. K. Vijay, Y. Lao, C. R. Lishka, J. Menon, J. Myers, S. A. Narayana, A. Procter, and T. J. Webb. Intel ngraph: An intermediate representation, compiler, and executor for deep learning. *CoRR*, abs/1801.08058, 2018.

[18] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. In *NeurIPS*, 2022.

[19] O. R. developers. Onnx runtime. `https://onnxruntime.ai/`, 2021.

[20] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[21] DHZS. tf-dropblock. `https://github.com/DHZS/tf-dropblock`.

[22] H. Face. Transformers. `https://github.com/huggingface/transformers`.

[23] W. Falcon et al. Pytorch lightning. *GitHub. Note: https://github.com/PyTorchLightning/pytorch-lightning*, 3, 2019.

[24] S. Fan, Y. Rong, C. Meng, Z. Cao, S. Wang, Z. Zheng, C. Wu, G. Long, J. Yang, L. Xia, et al. Dapple: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 431–445, 2021.

[25] W. Fedus, B. Zoph, and N. Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *CoRR*, abs/2101.03961, 2021.

[26] S. Feng, B. Hou, H. Jin, W. Lin, J. Shao, R. Lai, Z. Ye, L. Zheng, C. H. Yu, Y. Yu, and T. Chen. Tensorir: An abstraction for automatic tensorized program optimization. In *ASPLOS (2)*, pages 804–817. ACM, 2023.

[27] R. Frostig, M. J. Johnson, and C. Leary. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning*, 4(9), 2018.

[28] A. Griewank and A. Walther. Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software (TOMS)*, 26(1):19–45, 2000.

[29] C. R. Harris, K. J. Millman, S. J. Van Der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, et al. Array programming with numpy. *Nature*, 585(7825):357–362, 2020.

[30] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *CVPR*, pages 770–778. IEEE Computer Society, 2016.

[31] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, 1997.

[32] C. Huang, G. Jin, and J. Li. Swapadvisor: Pushing deep learning beyond the GPU memory limit via smart swapping. In *ASPLOS*, pages 1341–1355. ACM, 2020.

[33] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[34] P. Jain, A. Jain, A. Nrusimha, A. Gholami, P. Abbeel, K. Keutzer, I. Stoica, and J. Gonzalez. Checkmate: Breaking the memory wall with optimal tensor rematerialization. In *MLSys*. mlsys.org, 2020.

[35] A. Jangda, J. Huang, G. Liu, A. H. N. Sabet, S. Maleki, Y. Miao, M. Musuvathi, T. Mytkowicz, and O. Saarikivi. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 402–416, 2022.

[36] E. Jeong, S. Cho, G.-I. Yu, J. S. Jeong, D. Shin, and B.-G. Chun. Janus: Fast and flexible deep learning via symbolic graph execution of imperative programs. In *NSDI*, pages 453–468, 2019.

[37] X. Jia, L. Jiang, A. Wang, W. Xiao, Z. Shi, J. Zhang, X. Li, L. Chen, Y. Li, Z. Zheng, et al. Whale: Efficient giant model training over heterogeneous {GPUs}. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 673–688, 2022.

[38] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678, 2014.

[39] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *SOSP*, pages 47–62. ACM, 2019.

[40] Z. Jia, M. Zaharia, and A. Aiken. Beyond data and model parallelism for deep neural networks. *Proceedings of Machine Learning and Systems*, 1:1–13, 2019.

[41] N. P. Jouppi, C. Young, N. Patil, D. A. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. In *ISCA*, pages 1–12. ACM, 2017.

[42] W. Jung, T. T. Dao, and J. Lee. Deepcuts: a deep learning optimization framework for versatile GPU workloads. In *PLDI*, pages 190–205. ACM, 2021.

[43] C. Karakus, R. Huilgol, F. Wu, A. Subramanian, C. Daniel, D. Cavdar, T. Xu, H. Chen, A. Rahnama, and L. Quintela. Amazon sagemaker model

parallelism: A general and flexible framework for large model training. *arXiv preprint arXiv:2111.05972*, 2021.

[44] Kevin-Yang. MusicTransformer-tensorflow2.0. `https://github.com/jason9693/MusicTransformer-tensorflow2.0`.

[45] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[46] M. Kirisame, S. Lyubomirsky, A. Haan, J. Brennan, M. He, J. Roesch, T. Chen, and Z. Tatlock. Dynamic tensor rematerialization. In *ICLR*. OpenReview.net, 2021.

[47] M. Kirisame, S. Lyubomirsky, A. Haan, J. Brennan, M. He, J. Roesch, T. Chen, and Z. Tatlock. Dynamic tensor rematerialization. In *ICLR*. OpenReview.net, 2021.

[48] V. Korthikanti, J. Casper, S. Lym, L. McAfee, M. Andersch, M. Shoeybi, and B. Catanzaro. Reducing activation recomputation in large transformer models. *arXiv preprint arXiv:2205.05198*, 2022.

[49] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, pages 1106–1114, 2012.

[50] J. Kuen. Stochastic Downsampling for Cost-Adjustable Inference and Improved Regularization in Convolutional Networks (SDPoint). `https://github.com/xternalz/SDPoint`.

[51] W. Kwon, G. Yu, E. Jeong, and B. Chun. Nimble: Lightweight and parallel GPU task scheduling for deep learning. In *NeurIPS*, 2020.

[52] kyzhouhzau. NLPGNN. `https://github.com/kyzhouhzau/NLPGNN`.

[53] kyzhouhzau. NLPGNN. `https://github.com/kyzhouhzau/NLPGNN`.

[54] C. Lattner, J. A. Pienaar, M. Amini, U. Bondhugula, R. Riddle, A. Cohen, T. Shpeisman, A. Davis, N. Vasilache, and O. Zinenko. MLIR: A compiler infrastructure for the end of moore's law. *CoRR*, abs/2002.11054, 2020.

[55] D. Lepikhin, H. Lee, Y. Xu, D. Chen, O. Firat, Y. Huang, M. Krikun, N. Shazeer, and Z. Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

[56] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B. Su. Scaling distributed machine learning with the parameter server. In *OSDI*, pages 583–598. USENIX Association, 2014.

[57] S. Li and T. Hoefler. Chimera: efficiently training large-scale neural networks with bidirectional pipelines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[58] S. Li, F. Xue, C. Baranwal, Y. Li, and Y. You. Sequence parallelism: Long sequence training from system perspective. *arXiv e-prints*, pages arXiv–2105, 2021.

[59] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

[60] Z. Li, S. Zhuang, S. Guo, D. Zhuo, H. Zhang, D. Song, and I. Stoica. Terapipe: Token-level pipeline parallelism for training large-scale language models. In *International Conference on Machine Learning*, pages 6543–6552. PMLR, 2021.

[61] H. Lim, Y. Kim, S. Yun, J. Shin, and D. Han. Tspipe: Learn from teacher faster with pipelines. In *ICML*, volume 162 of *Proceedings of Machine Learning Research*, pages 13302–13312. PMLR, 2022.

[62] Y. Liu, Y. Wang, R. Yu, M. Li, V. Sharma, and Y. Wang. Optimizing CNN model inference on cpus. In *USENIX Annual Technical Conference*, pages 1025–1040. USENIX Association, 2019.

[63] L. Ma, Z. Xie, Z. Yang, J. Xue, Y. Miao, W. Cui, W. Hu, F. Yang, L. Zhang, and L. Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *OSDI*, pages 881–897. USENIX Association, 2020.

[64] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

[65] A. Mirhoseini, A. Goldie, H. Pham, B. Steiner, Q. V. Le, and J. Dean. A hierarchical model for device placement. In *ICLR (Poster)*. OpenReview.net, 2018.

[66] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean. Device placement optimization with reinforcement learning. In *ICML*, volume 70 of *Proceedings of Machine Learning Research*, pages 2430–2439. PMLR, 2017.

[67] D. Moldovan, J. Decker, F. Wang, A. Johnson, B. Lee, Z. Nado, D. Sculley, T. Rompf, and A. B. Wiltschko. Autograph: Imperative-style coding with graph-based performance. *Proceedings of Machine Learning and Systems*, 1:389–405, 2019.

[68] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[69] D. Narayanan, A. Phanishayee, K. Shi, X. Chen, and M. Zaharia. Memory-efficient pipeline-parallel dnn training. In *International Conference on Machine Learning*, pages 7937–7947. PMLR, 2021.

[70] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. 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.

[71] G. Neubig, C. Dyer, Y. Goldberg, A. Matthews, W. Ammar, A. Anastasopoulos, M. Ballesteros, D. Chiang, D. Clothiaux, T. Cohn, et al. Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980*, 2017.

[72] NVIDIA. cuBLAS: the CUDA Basic Linear Algebra Subroutine library. `https://docs.nvidia.com/cuda/cublas/index.html`.

[73] NVIDIA. CUTLASS: CUDA template abstractions for implementing high-performance matrix-matrix multiplication, note = `https://github.com/nvidia/cutlass`.

[74] NVIDIA. NVIDIA TensorRT: Programmable Inference Accelerator. `https://github.com/openxla/xla`.

[75] NVIDIA. NVIDIA TensorRT: Programmable Inference Accelerator. `https://developer.nvidia.com/tensorrt`.

[76] OpenAI. GPT-4 technical report. *CoRR*, abs/2303.08774, 2023.

[77] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, et al. Training language models to follow instructions with human feedback. *arXiv preprint arXiv:2203.02155*, 2022.

[78] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[79] X. Peng, X. Shi, H. Dai, H. Jin, W. Ma, Q. Xiong, F. Yang, and X. Qian. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 891–905, 2020.

[80] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo. A generic communication scheduler for distributed DNN training acceleration. In *SOSP*, pages 16–29. ACM, 2019.

[81] B. Pudipeddi, M. Mesmakhosroshahi, J. Xi, and S. Bharadwaj. Training large neural networks with constant memory using a new execution algorithm. *arXiv preprint arXiv:2002.05645*, 2020.

[82] Python Software Foundation. Pep 634: Structural pattern matching. `https://docs.python.org/3.10/whatsnew/3.10.html#pep-634-structural-pattern-matching`.

[83] PyTorch. torch.jit.trace. `https://pytorch.org/docs/1.8.0/generated/torch.jit.trace.html`.

[84] PyTorch. TorchScript. `https://pytorch.org/docs/1.8.0/jit.html`.

[85] PyTorch. TorchScript Language Reference. `https://pytorch.org/docs/1.8.0/jit_language_reference.html#language-reference`.

[86] M. N. Rabe and C. Staats. Self-attention does not need o(n^2) memory. *CoRR*, abs/2112.05682, 2021.

[87] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, G. Krueger, and I. Sutskever. Learning transferable visual models from natural language supervision. In *ICML*, volume 139 of *Proceedings of Machine Learning Research*, pages 8748–8763. PMLR, 2021.

[88] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, P. J. Liu, et al. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21(140):1–67, 2020.

[89] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. P. Amarasinghe. Halide: a language and compiler for optimizing parallelism,

locality, and recomputation in image processing pipelines. In *PLDI*, pages 519–530. ACM, 2013.

[90] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He. Zero: Memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

[91] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[92] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. 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.

[93] J. Reed, Z. DeVito, H. He, A. Ussery, and J. Ansel. torch. fx: Practical program capture and transformation for deep learning in python. *Proceedings of Machine Learning and Systems*, 4:638–651, 2022.

[94] J. Ren, S. Rajbhandari, R. Y. Aminabadi, O. Ruwase, S. Yang, M. Zhang, D. Li, and Y. He. {ZeRO-Offload}: Democratizing {Billion-Scale} model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 551–564, 2021.

[95] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[96] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer. High-resolution image synthesis with latent diffusion models. In *CVPR*, pages 10674–10685. IEEE, 2022.

[97] N. Shazeer, Y. Cheng, N. Parmar, D. Tran, A. Vaswani, P. Koanantakool, P. Hawkins, H. Lee, M. Hong, C. Young, et al. Mesh-tensorflow: Deep learning for supercomputers. *Advances in neural information processing systems*, 31, 2018.

[98] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[99] A. Singh. gpt-2-tensorflow2.0. `https://github.com/akanyaani/gpt-2-tensorflow2.0`.

[100] M. Sivathanu, T. Chugh, S. S. Singapuram, and L. Zhou. Astra: Exploiting predictability to optimize deep learning. In *ASPLOS*, pages 909–923. ACM, 2019.

[101] S. Smith, M. Patwary, B. Norick, P. LeGresley, S. Rajbhandari, J. Casper, Z. Liu, S. Prabhumoye, G. Zerveas, V. Korthikanti, et al. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. *arXiv preprint arXiv:2201.11990*, 2022.

[102] A. Suhan, D. Libenzi, A. Zhang, P. Schuh, B. Saeta, J. Y. Sohn, and D. Shabalin. Lazytensor: combining eager execution with domain-specific compilers. *CoRR*, abs/2102.13267, 2021.

[103] J. M. Tarnawski, D. Narayanan, and A. Phanishayee. Piper: Multidimensional planner for dnn parallelization. *Advances in Neural Information Processing Systems*, 34:24829–24840, 2021.

[104] TensorFlow. Autograph Limitation Document. `https://github.com/tensorflow/tensorflow/blob/r2.4/tensorflow/python/autograph/g3doc/reference/limitations.md`.

[105] TensorFlow. Deep Convolutional Generative Adversarial Network Tutorial. `https://www.tensorflow.org/tutorials/generative/dcgan`.

[106] TensorFlow. TensorFlow Model Garden. `https://github.com/tensorflow/models`.

[107] TensorFlow. tf.function. `https://www.tensorflow.org/versions/r2.4/api_docs/python/tf/function`.

[108] TensorFlow. TFRT: A New TensorFlow Runtime. `https://github.com/tensorflow/runtime`.

[109] R. Thoppilan, D. D. Freitas, J. Hall, N. Shazeer, A. Kulshreshtha, H. Cheng, A. Jin, T. Bos, L. Baker, Y. Du, Y. Li, H. Lee, H. S. Zheng, A. Ghafouri, M. Menegali, Y. Huang, M. Krikun, D. Lepikhin, J. Qin, D. Chen, Y. Xu, Z. Chen, A. Roberts, M. Bosma, Y. Zhou, C. Chang, I. Krivokon, W. Rusch, M. Pickett, K. S. Meier-Hellstern, M. R. Morris, T. Doshi, R. D. Santos, T. Duke, J. Soraker, B. Zevenbergen, V. Prabhakaran, M. Diaz, B. Hutchinson, K. Olson, A. Molina, E. Hoffman-John, J. Lee, L. Aroyo, R. Rajakumar, A. Butryna, M. Lamm, V. Kuzmina, J. Fenton, A. Cohen, R. Bernstein, R. Kurzweil, B. Aguera-Arcas, C. Cui, M. Croak, E. H. Chi, and Q. Le. Lamda: Language models for dialog applications. *CoRR*, abs/2201.08239, 2022.

[110] P. Tillet, H. Kung, and D. D. Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *MAPL@PLDI*, pages 10–19. ACM, 2019.

[111] S. Tokui, R. Okuta, T. Akiba, Y. Niitani, T. Ogawa, S. Saito, S. Suzuki, K. Uenishi, B. Vogel, and H. Yamazaki Vincent. Chainer: A deep learning framework for accelerating the research cycle. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2002–2011, 2019.

[112] C. Unger, Z. Jia, W. Wu, S. Lin, M. Baines, C. E. Q. Narvaez, V. Ramakrishnaiah, N. Prajapati, P. McCormick, J. Mohd-Yusof, et al. Unity: Accelerating {DNN} training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 267–284, 2022.

[113] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR*, abs/1802.04730, 2018.

[114] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[115] M. Verma. Revisiting linformer with a modified self-attention with linear complexity. *CoRR*, abs/2101.10277, 2021.

[116] Viredery. tf-eager-fasterrcnn. `https://github.com/Viredery/tf-eager-fasterrcnn`.

[117] L. Wang, J. Ye, Y. Zhao, W. Wu, A. Li, S. L. Song, Z. Xu, and T. Kraska. Superneurons: Dynamic gpu memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 41–53, 2018.

[118] M. Wang, C.-c. Huang, and J. Li. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–17, 2019.

[119] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, and J. Brew. Huggingface's transformers: State-of-the-art natural language processing. *CoRR*, abs/1910.03771, 2019.

[120] Y. Xu, H. Lee, D. Chen, B. Hechtman, Y. Huang, R. Joshi, M. Krikun, D. Lepikhin, A. Ly, M. Maggioni, et al. Gspmd: general and scalable parallelization for ml computation graphs. *arXiv preprint arXiv:2105.04663*, 2021.

[121] B. Yang, J. Zhang, J. Li, C. Ré, C. Aberger, and C. De Sa. Pipemare: Asynchronous pipeline parallel dnn training. *Proceedings of Machine Learning and Systems*, 3:269–296, 2021.

[122] T. W. H. W. Yanjun Ma, Dianhai Yu. Paddlepaddle: An open-source deep learning platform from industrial practice. *Frontiers of Data and Domputing*, 1(1):105, 2019.

[123] J. Yuan, X. Li, C. Cheng, J. Liu, R. Guo, S. Cai, C. Yao, F. Yang, X. Yi, C. Wu, H. Zhang, and J. Zhao. Oneflow: Redesign the distributed deep learning framework from scratch. *CoRR*, abs/2110.15032, 2021.

[124] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. T. Diab, X. Li, X. V. Lin, T. Mihaylov, M. Ott, S. Shleifer, K. Shuster, D. Simig, P. S. Koura, A. Sridhar, T. Wang, and L. Zettlemoyer. OPT: open pre-trained transformer language models. *CoRR*, abs/2205.01068, 2022.

[125] Z. Zhang. yolov3-tf2. `https://github.com/zzh8829/yolov3-tf2`.

[126] J. Zhao, B. Li, W. Nie, Z. Geng, R. Zhang, X. Gao, B. Cheng, C. Wu, Y. Cheng, Z. Li, P. Di, K. Zhang, and X. Jin. AKG: automatic kernel generation for neural processing units using polyhedral transformations. In *PLDI*, pages 1233–1248. ACM, 2021.

[127] Y. Zhao, A. Gu, R. Varma, L. Luo, C.-C. Huang, M. Xu, L. Wright, H. Shojanazeri, M. Ott, S. Shleifer, et al. Pytorch fsdp: Experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277*, 2023.

[128] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, J. E. Gonzalez, and I. Stoica. Ansor: Generating high-performance tensor programs for deep learning. In *OSDI*, pages 863–879. USENIX Association, 2020.

[129] L. Zheng, Z. Li, H. Zhang, Y. Zhuang, Z. Chen, Y. Huang, Y. Wang, Y. Xu, D. Zhuo, J. E. Gonzalez, et al. Alpa: Automating inter-and intra-operator parallelism for distributed deep learning. *arXiv preprint arXiv:2201.12023*, 2022.

[130] S. Zheng, Y. Liang, S. Wang, R. Chen, and K. Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor

computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 859–873, 2020.

[131] Z. Zheng, P. Zhao, G. Long, F. Zhu, K. Zhu, W. Zhao, L. Diao, J. Yang, and W. Lin. Fusionstitching: Boosting memory intensive computations for deep learning workloads. *CoRR*, abs/2009.10924, 2020.

[132] Y. Zhou, S. Roy, A. Abdolrashidi, D. Wong, P. C. Ma, Q. Xu, H. Liu, M. P. Phothilimtha, S. Wang, A. Goldie, A. Mirhoseini, and J. Laudon. Transferable graph optimizers for ML compilers. In *NeurIPS*, 2020.

[133] Y. Zhuang, H. Zhao, L. Zheng, Z. Li, E. P. Xing, Q. Ho, J. E. Gonzalez, I. Stoica, and H. Zhang. On optimizing the communication of model parallelism. *arXiv preprint arXiv:2211.05322*, 2022.

초록

딥러닝 모델에 대한 수요가 빠르게 증가하면서, 딥러닝 학습을 위한 소프트웨어 시스템들의 빠른 발전도 촉진하고 있다. 그러한 소프트웨어 시스템들은 딥러닝 모델들이 여러 딥러닝 가속기들의 계산 리소스를 최대로 이용하면서 학습할 수 있도록 상당히 완성도 있는 최적화를 지원한다. 하지만, 딥러닝 모델 구조가 다양해지고, 모델의 크기가 계속 증가하면서 그러한 최적화에 방해가 되는 요소들이 끊임없이 생겨나고 있다. 만약 그러한 요소들을 해결하지 못하면, 비효율적으로 딥러닝 학습을 수행하게 된다. 이 논문에서는, 이러한 비효율성의 종류와 원인을 분석하고, 이를 해결하는 두 가지 새로운 시스템을 소개한다.

첫 번째 시스템 테라는 (Terra) 명령형 (imperative) 수행 모델이 갖는 비효율적인 학습 속도를 해결한다. 테라는 명령형으로 딥러닝 프로그램을 수행을 하는 동시에 딥러닝 연산들을 분리하여 심볼릭 (symbolic) 수행한다. 그에 따라, 테라는 명령형 수행을 염두에 두고 작성된 딥러닝 프로그램에도 심볼릭 수행의 빠른 최적화를 적용할 수 있게 하고, 명령형 수행 대비 최대 1.73배 빠른 학습을 지원한다. 그 다음으로 소개하는 시스템인 비파이프는 (BPipe) 대규모 모델 학습에서 메모리 비효율성을 해결한다. 비파이프는 학습 중간 생성되는 활성화 텐서량의 균형을 맞추는 새로운 파이프라인 병렬 학습 방법을 제시한다. 비파이프를 사용하면, 분산학습에서 모든 딥러닝 가속기들이 비슷한 양의 메모리를 사용하도록 만들어서, 최대 2.17배 만큼 빠른 학습을 수행할 수 있다.

주요어: 딥러닝 프레임워크, 대규모 언어 모델 학습, 분산학습

학번: 2019-25320