Chakra: Advancing Performance Benchmarking and Co-design using Standardized Execution Traces

Srinivas Sridharan[1*], Taekyung Heo[1, 2*], Louis Feng[1], Zhaodong Wang[1], Matt Bergeron[1], Wenyin Fu[1], Shengbao Zheng[1], Brian Coutinho[1], Saeed Rashidi[2], Changhai Man[2], Tushar Krishna[2]
[1]Meta
[2]Georgia Institute of Technology

ABSTRACT

Benchmarking and co-design are essential for driving optimizations and innovation around ML models, ML software, and next-generation hardware. Full workload benchmarks, e.g. MLPerf, play an essential role in enabling fair comparison across different software and hardware stacks especially once systems are fully designed and deployed. However, the pace of AI innovation demands a more agile methodology to benchmark creation and usage by simulators and emulators for future system co-design. We propose Chakra[1], an open graph schema for standardizing workload specification capturing key operations and dependencies, also known as Execution Trace (ET). In addition, we propose a complementary set of tools/capabilities to enable collection, generation, and adoption of Chakra ETs by a wide range of simulators, emulators, and benchmarks. For instance, we use generative AI models to learn latent statistical properties across thousands of Chakra ETs and use these models to synthesize Chakra ETs. These synthetic ETs can obfuscate key proprietary information and also target future what-if scenarios. As an example, we demonstrate an end-to-end proof-of-concept that converts PyTorch ETs to Chakra ETs and uses this to drive an open-source training system simulator (ASTRA-sim). Our end-goal is to build a vibrant industry-wide ecosystem of agile benchmarks and tools to drive future AI system co-design.

ACM Reference Format:
Srinivas Sridharan[1*], Taekyung Heo[1, 2*], Louis Feng[1], Zhaodong Wang[1], Matt Bergeron[1], Wenyin Fu[1], Shengbao Zheng[1], Brian Coutinho[1], Saeed Rashidi[2], Changhai Man[2], Tushar Krishna[2]. 2023. Chakra: Advancing Performance Benchmarking and Co-design using Standardized Execution Traces. In *Proceedings of Workshop on Benchmarking Machine Learning Workloads on Emerging Hardware (MLBench'23).* ACM, New York, NY, USA, 9 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Modern deep learning systems heavily rely on distributed training over high-performance *Neural Processing Unit* (NPUs) (i.e., GPU or TPU)-based hardware platforms (e.g., Meta ZionEX, Google CloudTPU, NVIDIA DGX). Distributing the training task involves splitting the datasets (data parallel) or splitting the model weights (model parallel) or splitting the model layers (pipeline parallel) or hybrid combinations (3D/4D parallel). The optimal parallelism strategy is a topic of active research [2, 7].

There is an agreement in the community that benchmarking and HW-SW co-design are crucial for developing next-generation AI/ML platforms. Full workload benchmarks, e.g. MLPerf, play an essential role in enabling fair comparison across different software and hardware stacks especially once systems are fully designed and deployed. However, the pace of AI innovation demands a more agile methodology for benchmark creation and usage by simulators and emulators for future system co-design.

Furthermore, state-of-the-art AI/ML models and custom AI/ML accelerators (NPUs) are often designed by different organizations (e.g., DLRM designed by Meta running on NVIDIA GPUs) – and these organizations (especially industry) cannot disclose full details due to proprietary intellectual properties (IPs) and technologies. As a result, hyperscalar cloud providers today end up sharing spreadsheets with model parameters under NDA with some vendors, but it is hard to derive/reproduce exact workload details; while the majority of the hardware vendors (and academic researchers) have to derive parameters from MLPerf or other benchmarks, often over-optimize a few use cases and ignoring Amdahl's law.

To overcome these challenges, this paper proposes Chakra, an infrastructure for enabling benchmarking of ML models on *future* systems and performing co-design-space exploration. The key enabler of Chakra is the idea of *execution traces (ET)*. Analogous to instruction or memory traces [8] for optimizing compute and memory architectures, ETs encode critical information related to compute and communication operator dimensions and dependencies while not revealing model or dataset details. SW companies can share ET for their internal workloads of interest to HW vendors who can in-turn estimate and optimize performance on next-generation NPU platforms using their proprietary HW model/simulator(s). The feedback can help SW companies identify optimized compute and network configurations for their training tasks. The availability of ETs can also help academic researchers and HW startups participate in co-design for real production ML workloads.

The information required by ET is in fact readily available during execution of a ML workload using a framework like PyTorch/TensorFlow, but not explicitly exposed today. This paper presents the key components of the Chakra infrastructure - (i) an ET schema, called `Chakra ET` that captures relevant execution and dependency

[1]https://github.com/chakra-et/chakra

MLBench'23, June 8, 2023, Miami, FL
© 2023 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

[*]These authors contributed equally to this work

information[*], (ii) open-source toolchains such as visualizers, converter, and simulators, and (iii) generative AI model for synthesizing execution traces.

2 CHAKRA OVERVIEW

In this section, we overview Chakra, a performance modeling framework for distributed ML workloads. Comprising various concepts and tools, Chakra enables users to estimate the execution time and resource usage of a distributed ML task for a given system configuration. Figure 1 illustrates the key components of the Chakra infrastructure, with execution traces (ETs) serving as the central element. These ETs facilitate the exchange of ML execution traces without revealing model specifics. Generated by ML frameworks, ETs are converted to a common format, `Chakra ET`, which are subsequently fed into simulators to estimate the performance of ML workloads. In this section, we provide an overview of the Chakra project and its main components.

2.1 Motivation

The importance of HW-SW codesign in designing distributed ML systems is well-known. Despite its importance, the necessary infrastructure for effective implementation remains insufficient. Three primary challenges hinder progress in this area. Firstly, the absence of a unified schema for exchanging execution traces of ML models poses a significant obstacle. Although numerous ML frameworks and languages exist, their primary focus has been on representing and optimizing ML models, with performance modeling receiving less attention. Consequently, no standard schema for encoding execution information has been developed. Secondly, the lack of toolchains for performance modeling is another challenge. Comprehensive toolchains are crucial for identifying bottlenecks and debugging issues efficiently. Lastly, current methodologies lack the ability to synthesize execution traces. Synthesizing traces is vital for several reasons, including concerns over sharing intellectual property when exchanging execution traces between different organizations. By generating synthetic traces that retain the characteristics of real-world data while obscuring sensitive details, companies can share information without compromising their proprietary knowledge. Furthermore, synthesized execution traces play a critical role in performance projection, as future systems will likely have varying numbers of NPUs and connectivity configurations.

2.2 Schema

In Chakra, execution traces are in the Chakra schema, which is defined as a common format for exchanging execution traces for performance modeling purposes. The need for a standardized schema arises as ML frameworks often provide methods to collect execution traces of ML models, but these traces are presented in varying formats, which obstructs the exchange of information between different organizations. To address this issue, the Chakra schema is designed based on the requirements of various organizations, including universities and companies. Moreover, the schema facilitates the collection of traces at different stages, such as pre-execution and post-execution, allowing for the gathering of traces that are either specific to a particular system design or not.

[*]We plan to standardize this via MLCommons, OCP, or other existing standards organization in consultation with other hyperscalars and hardware vendors.

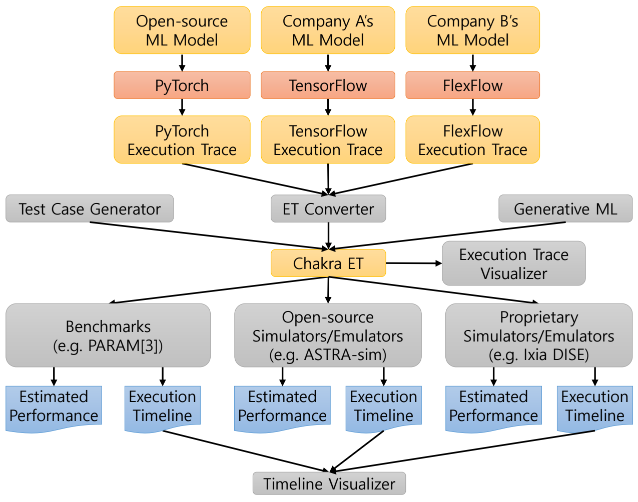


Figure 1: Chakra infrastructure overview.

2.3 Chakra ET Collection and Synthesis

Execution traces can be collected from real-world ML frameworks or synthetic trace generators, each with its own pros and cons. ML frameworks offer the most straightforward method for collecting execution traces, as they run ML models directly. We extended PyTorch to support trace collection with the Execution Graph Observer [1], which does not require intrusive modifications to ML models. Simply enabling the Execution Graph Observer before running a model yields traces that closely reflect real-world execution.

However, this approach has limitations as the resulting execution traces are closely tied to real-world systems. Factors such as the number of NPUs, computation time, and network delays are tightly coupled with the actual system. Synthetic trace generators can overcome these limitations. In our study, we designed a generative ML model trained with production traces to capture the characteristics of ML traces, such as the relationships between nodes in a trace and their fields. This model can be used to simulate traces of various ML models by varying the number of NPUs. A key advantage of the generative ML model is its ability to obfuscate traces, which is critical for sharing ML traces while protecting intellectual property. Obfuscation is a significant concern when sharing ML traces, and the generative ML model can address this issue by conveying the essential characteristics of the trace while maintaining privacy.

2.4 Open-source Tools

Chakra offers a range of open-source tools to help users modify and better understand the execution traces. These tools include the execution trace converter, execution trace visualizer, timeline visualizer, test case generator, and execution trace feeder, all of which are currently available on our GitHub repository[*] and under active development. The execution trace converter is responsible for converting execution traces in various schemas to the Chakra schema. The execution trace visualizer allows users to visualize the dependencies between nodes in a trace. Meanwhile, the timeline visualizer provides a visualization of the execution of nodes when a trace is simulated in a simulator. The test case generator lets users generate arbitrary execution traces by offering libraries to describe traces. The execution trace feeder is a library that other simulators can use to parse traces and feed them into the simulator. Further details will be presented in Section 5.

2.5 Use Cases

Chakra serves a variety of purposes, including HW-SW co-design, benchmarking, and performance projection. The proposed Chakra schema serves as a unified format for exchanging execution traces, significantly simplifying the process of sharing these traces. Additionally, the generative AI model enables users to produce representative traces based on existing ones. As a result, NPUs or training systems can be optimized using these traces. Furthermore, Chakra can be employed for performance projection, allowing users to estimate the potential performance improvements resulting from investments by modeling systems with varying compute engines or network topologies. A key advantage of Chakra is that it allows companies to use their own internal simulators or toolchains, provided they follow the common format.

3 CHAKRA SCHEMA

This section provides a detailed description of the Chakra schema. In this section, we present the goal of the schema and its design. Additionally, we discuss the important design choices made during the development of the Chakra schema.

3.1 Design Requirements

ML tasks are often presented as graphs, with frameworks such as TensorFlow and PyTorch constructing computational graphs internally to represent model execution [5]. In this study, we introduce *an execution trace* (ET) as a graph that captures the execution details of an ML task. An ET is a trace of ML execution that includes details such as memory accesses, computational load, network communication, and parallelization strategies. While ETs are a useful way to represent an ML task, the structures and metadata of generated ETs may differ depending on the ML framework used. To ensure that the Chakra framework can be used across different ML frameworks, we propose a standard ET schema for performance modeling called Chakra execution trace (`Chakra ET`).

The Chakra schema was designed to meet the diverse requirements of various teams and organizations. To achieve this goal, we had meetings with many universities and companies to gather input from different teams. We present the requirements that were incorporated into the schema. The first requirement is that the schema should be minimal yet extensible. This is a common requirement across all organizations because they do not want to use up resources on non-used fields, but they also want to extend the schema for their specific purposes. The second requirement is that the schema should be able to model various aspects of ML execution from a performance modeling perspective. While there are many languages and representations designed for presenting ML models, there is no common schema for modeling the performance of ML execution. Therefore, performance modeling is a critical aspect of schema design. The schema should be able to model memory access, computation, and network communication. Additionally, the trace should have dependencies between nodes because many operations have dependencies. The third requirement is that the schema should be able to present execution traces at different stages of execution. If the resulting trace is too tightly coupled to the real system execution, it cannot be used for performance projection or estimation. Therefore, the schema should be able to present execution traces at any level of ML execution.

[*] https://github.com/chakra-et/chakra

Field Name	Data Type	Description
id	required uint64	Node ID
name	required string	Node name
type	required enum	Node type
parent	repeated uint64	Parent node IDs
attribute	repeated AttributeProto	Any additional fields

Table 1: Chakra node schema.

Field Name	Data Type	Description
name	required string	Name of this attribute
type	required enum	Type of this attribute
doc_string	required string	Description of this attribute
f	optional float	Float value
i	optional int64	Integer value
s	optional string	String value
floats	repeated float	Repeated float values
ints	repeated int64	Repeated integer values
strings	repeated string	Repeated string values

Table 2: Chakra AttributeProto schema.

3.2 Schema

This subsection provides an in-depth look at the Chakra schema, which is presented in Table 1 and Table 2. Table 1 outlines the Chakra node schema, while Table 2 presents the AttributeProto schema. During the design phase, we aimed to create a schema with minimal required fields while making it highly extensible. As demonstrated in Table 1, the schema only requires a few fields to represent a node, such as id, name, type, parent, and attribute. The id field is used to uniquely identify a node, while the name field is used for naming nodes. The nodes in Chakra currently represent compute, memory and communication/networking operations. The type field, defined as an enum, can be one of the following: INVALID, MEM_LOAD, MEM_STORE, COMP, COMM_SEND, COMM_RECV, or COMM_COLL[*]. INVALID works as an initial value of the type field, and it is used to ignore non-critical nodes. Memory types are used for tracing and modeling memory systems, and communication types are used for network systems. All additional metadata to model computation, memory access, and network are encoded in

[*] SEND and RECV are used for pt-to-pt messages, while COLL for collective messages.

the attribute field. The parent field is a repeated field that holds the IDs of parent nodes to encode dependencies between nodes. Finally, the attribute field is a repeated field that significantly improves the schema's extensibility. This idea was inspired by the ONNX schema. Table 2 shows the schema for AttributeProto, which is essentially a key-value pair. AttributeProto supports six data types that can be identified using the type field: float, int, string, and their repetitions. It should be emphasized that this schema satisfies all of the requirements introduced in Section 3.1.

Please note that this schema satisfies all of the requirements introduced in Section 3.1. First, the schema is designed to be minimal and extensible. The Chakra node schema has critical fields such as id, name, type, parent, and attribute. All other fields can be presented as in the attribute field. However, this design necessitates complex execution trace collection and parsing codes to read and interpret all attribute fields. Second, the schema can model the performance of ML execution. As the type field suggests, nodes in a trace represent memory access, computation, and communication. Their dependencies are presented with the parent field, and various compute engines and networks can be modeled by adding more metadata in the attribute field. Third, the schema can present traces collected at any level because the attribute field is extensible.

3.3 Discussion

This subsection delves into the design choices made and their associated implications.

3.3.1 Pre-execution vs. post-execution. Determining the appropriate level for trace collection is a crucial design choice. Generally, trace collection levels can be classified as either pre-execution or post-execution. Pre-execution refers to a stage where an ML model has not yet been explicitly optimized or tied to specific system configurations, such as compute engine type, memory bandwidth, or network topology. Collecting a trace at the pre-execution stage allows for performance projection across various systems, as it is not tethered to a particular configuration. Conversely, the post-execution stage involves optimizing and executing the ML model on an actual system. Traces gathered during this stage are tightly linked to the real system on which the model is run. Although these traces accurately represent real-world phenomena, their utility for performance projection is limited due to the varying optimization and parallelization strategies required by different systems. The Chakra schema is designed to accommodate any execution stage. However, how to collect traces from a real system without being tightly bound to it remains an open question.

3.3.2 Multi-node representation. In the current Chakra implementation, it is assumed that each NPU possesses a corresponding trace. Given N NPUs, N distinct traces are generated. This design choice offers the advantage of simplifying trace collection and replay, as each NPU does not need to concern itself with the traces or execution of other NPUs. Nonetheless, it constrains the potential for global optimization, wherein a global scheduler could dynamically allocate work to each NPU or redistribute tasks from one NPU to another. Although the Chakra schema does not inherently restrict such an approach, it essentially operates under the assumption that each NPU maintains an individual trace.

3.3.3 Execution graphs vs. execution traces. Execution graphs and execution traces are closely related concepts, both represented as graphs, but they serve distinct purposes. Execution graphs are designed to facilitate precise model execution and performance optimization within machine learning frameworks such as PyTorch, TensorFlow, and ONNX. In contrast, execution traces do not focus on the exact execution of models; rather, they concentrate on replaying, simulating, and emulating processes. The primary objective of execution traces is to assess performance and coverage, ideally operating in a framework-agnostic manner.

4 ET COLLECTION AND SYNTHESIS

Collecting all production traces data on a regular cadence is feasible with the tooling support provided by Chakra. However, due to the massive amount of data, it is not scalable or effective to replay all of them for benchmarking. Additionally, sharing these traces may not be advised since it would be possible to reconstruct the original model. This could result in proprietary model information being shared widely. Therefore, automatically discovering a representative set of traces is crucial to complete the whole system. To address this issue, we propose a methodology for generating representative traces using generative machine learning models based on production trace data.

Our approach enables efficient generation of realistic-looking production and futuristic traces, which can be used for data-driven decision making. Furthermore, it provides representative traces as the single source of truth that can be shared internally and externally given the stochastic nature of the generative models, which naturally provides an obfuscation of the production data. We start our modeling work from analyzing the communication part of the trace, the same framework can be easily extended to full trace.

The communication part of the traces contains communication collectives across a set of GPUs (ranks). There are a few hard constraints on the traces within the same process group to ensure the replay system can run these traces without raising errors. To better address these constraints, we propose a concept of "master trace", which is a lossless compression of all N traces of all ranks taking all constraints into consideration. We can construct a master trace from N traces of all ranks, and reconstruct N traces from this master trace without losing any information. The master trace helps us to learn a distribution and synthesize traces without worrying about violating the constraints of the replay system.

Based on the master trace data, we developed a hierarchical ML model comprising several generators that ensure the synthesized traces follow the expected data distributions. See Figure 2 for an overview of this model. There are a couple of advantages for the hierarchical model that facilitates model research and experimentation. It is modularized and flexible. We can substitute the individual generator parts with various algorithms. We can easily extend the model with additional generation stages when needed. This facilitates model research and experimentation. It is also interpretable and easy for debugging. The model is conceptually easy to understand. Even when we use complex algorithms for the generators, we are able to evaluate performance at each stage and identify bottlenecks for improvements. This is a big advantage for model debugging and interpretation compared to a single blackbox model

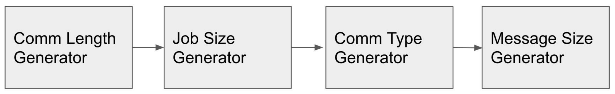


Figure 2: Generative AI model for ET synthesis.

where debugging is difficult. Moreover, the model of generators follow the actual param trace generation logic. It provides us data insights into the param trace generation stages, enabling deeper understanding of param trace patterns.

For network use cases, the composition of collective types and the message sizes are the key factors. The Comm Type Generator ensures that the generated collective types align with the collective composition of production traces. We have identified two major clusters of collective type compositions. The Message Size Generator ensures that the message size generated follows the distribution of each collective type. We developed Gaussian mixture models to fit the data and draw samples from the fitted distribution. The synthesized samples align relatively well with the true distribution. We have modeled the distributions of comm types, comm sequence length, GPU sizes, message sizes, autocorrelation between comm types and message sizes, and message splits across GPUs, as well as the dependencies between the generators. We have tested the synthesized traces using the replay system and found that they work correctly on our training cluster.

The generative models can be extended to generate traces for larger futuristic system scales and compute & communication traces, which will capture compute-communication overlap, dependency structure, etc. Overall, our generative AI models enables efficient generation of representative traces, which will be a valuable tool for optimizing distributed AI training and evaluating future hardware designs.

5 OPEN-SOURCE TOOLS

In this section, we introduce the open-source tools offered by the Chakra project, which are currently under active development. More tools can be added in the future.

5.1 Execution Trace Converter

While various ML frameworks allow users to collect execution traces of ML models, they are often in different formats, hindering the exchange of traces between different organizations. The ET converter's role is to translate ETs from various schemas into the Chakra schema. Currently, the converter supports PyTorch and FlexFlow [2], and considering that FlexFlow supports Keras, PyTorch, and ONNX, it can support most ML frameworks. TensorFlow support is planned for future work.

The ET converter has prior knowledge of the input execution trace schema and can extract Chakra fields from the input file. For instance, PyTorch execution traces are in the JSON format, and the converter uses the JSON parser to read the execution traces. In PyTorch, computation nodes have the duration field if computation time is encoded after collecting traces, allowing computation nodes to be identified. Communication nodes in PyTorch are identified

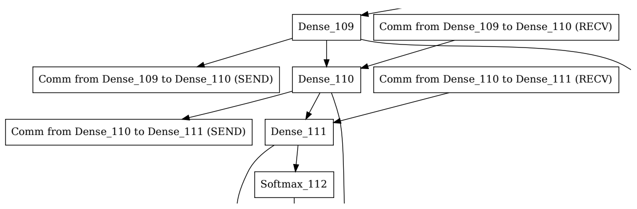

Dense_109
Comm from Dense_109 to Dense_110 (RECV)
Comm from Dense_109 to Dense_110 (SEND)
Dense_110
Comm from Dense_110 to Dense_111 (RECV)
Comm from Dense_110 to Dense_111 (SEND)
Dense_111
Softmax_112

Figure 3: Execution trace visualization example.

by the name of nodes, which is "record_param_comms". The communication type and size are determined by its child node, whose name starts with "nccl:". Unlike PyTorch traces, FlexFlow traces are in the graphviz format, and the FlexFlow codes needed to be slightly modified to identify node types and convey more metadata. Node types in FlexFlow are identified with the name field, and each node has corresponding metadata fields. One significant difference between FlexFlow and PyTorch is that collective communications in FlexFlow are broken down into multiple peer-to-peer data transfers. If the converter encounters uninterpretable nodes, they are classified as invalid nodes and ignored in other toolchains. It is important to note that in Chakra, each NPU generates a separate trace. Therefore, if an input file describes the behavior of all NPUs, the converter must split it into multiple pieces correctly.

5.2 Execution Trace Visualizer

The execution trace visualizer is designed to visualize execution traces in the Chakra schema. The visualizer's input file should conform to the Chakra schema, and it can output files in either the graphviz or PDF format. The visualizer is a helpful tool for researchers to understand the structure of execution traces. Moreover, it can be used for debugging simulators or emulators that take Chakra execution traces. By default, the visualizer encodes the names of tasks and dependencies between tasks. Users can easily modify the visualizer to encode additional metadata such as compute time and communication size. An example of a visualization from the execution trace visualizer is presented in Figure 3.

5.3 Execution Timeline Visualizer

The execution timeline visualizer presents task execution on each NPU in a timeline, taking a CSV file describing the task execution in a callback-based simulator. An example input format for the visualizer is presented in Snippet 1. The output timeline can be visualized with the trace event profiling tool in Chrome (chrome://tracing). Figure 4 shows a screenshot of the visualization result, where tasks running on each NPU are presented as bars on the timeline, with their start and finish times. Tasks on an NPU are grouped as a process, and are classified into several sub-groups depending on the task type. In this example, two NPUs are shown, and memory access, computation, and communication have TID 1, 2, and 3, respectively. This tool allows researchers to easily identify bottlenecks in distributed ML tasks. For instance, in NPU 1, communication time is not overlapped with computation, unlike in NPU 2. Therefore, the task in NPU 1 can be optimized by overlapping communication with computation.

```
issue ,[gpu_id],[curr_cycle],[node_id],[node_name]
callback ,[gpu_id],[curr_cycle],[node_id],[node_name]
issue ,[gpu_id],[curr_cycle],[node_id],[node_name]
issue ,[gpu_id],[curr_cycle],[node_id],[node_name]
callback ,[gpu_id],[curr_cycle],[node_id],[node_name]
callback ,[gpu_id],[curr_cycle],[node_id],[node_name]
...
```

Snippet 1: Execution timeline visualizer input format.

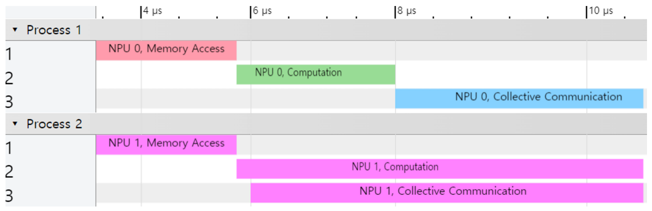


Figure 4: Execution timeline visualization example.

5.4 Test Case Generator

While execution traces are expressive and can effectively encode real-world ML traces, their limited readability poses challenges for implementing ET-based tools. The test case generator allows users to define arbitrary execution traces that can subsequently be used to generate test cases. Additionally, the test case generator allows manual modeling of ML models. The test trace generator offers an array of C++ classes for composing Chakra execution traces. Snippet 2 demonstrates how an execution trace can be defined within the tool. As illustrated in the code snippet, users can create a new trace and designate the dependencies between nodes. We also offer several default trace examples, encompassing data parallel, model parallel, and pipeline parallel instances for a synthetic ML model.

5.5 Trace Feeder for Simulators

We offer a trace feeder library that is designed for simulators. One of the use cases for Chakra is simulating distributed ML systems for performance projection or what-if analysis. We anticipate that numerous companies and research institutions will actively share traces in the Chakra schema and use them with their simulators, whether open-source such as ASTRA-sim [9, 14] or SST [10], or proprietary ones. The trace feeder library is implemented in C++, eliminating the need for users to repeatedly implement the parsing code. The trace feeder library significantly reduces users' engineering overhead. The public member functions of the trace feeder class are presented in Snippet 3. With these functions, users can create a trace feeder instance and retrieve nodes or resolve dependencies.

6 USE CASES

6.1 Replay Benchmarks

Traditional benchmark creation involves starting with choosing representative applications and adapting and/or simplifying their implementation to highlight a few specific behaviors. On the contrary, detailed operator and dependency information from Chakra ETs can be used to develop *replay benchmarks*. These benchmarks essentially replay the graph to mimic the application behavior. Chakra

```
et = new DependencyGraph(getFilename(..., npu_id));

node1 = et->addNode(ChakraProtoMsg::COMP_NODE);
node1->set_name("COMP_NODE");
node1->set_simulated_run_time(5);

node2 = et->addNode(ChakraProtoMsg::COMP_NODE);
node2->set_name("COMP_NODE");
node2->set_simulated_run_time(5);

et->assignDep(node1, node2);
```

Snippet 2: Test case generator codes.

```
void addNode(std::shared_ptr<EGFeederNode> node);
void removeNode(uint64_t node_id);
bool hasNodesToIssue();
std::shared_ptr<EGFeederNode> getNextIssuableNode();
void pushBackIssuableNode(uint64_t node_id);
std::shared_ptr<EGFeederNode> lookupNode(uint64_t node_id);
void freeChildrenNodes(uint64_t node_id);
```

Snippet 3: Trace feeder class public member functions.

ET replay benchmarks offer the following unique advantages. First, ET replay drastically reduces the benchmark software dependency (e.g. various libraries a full application usually comes with, many of them do not have a direct relationship to the performance evaluation, but rather a vestige of the original application's other business logic). This makes the resulting benchmark agile and easy to deploy. Second, creating/adapting an application to benchmarks requires a lot of efforts; hence, creating benchmarks directly from ET collected during the normal fleet operation can save a tremendous amount of work. Lastly, because it is possible to build an automatic benchmark creation pipeline continuously, either sourcing from the fleet workloads or through the fore-mentioned generative AI techniques, we can keep our benchmark suites up-to-date at scale.

PARAM Comms Replay benchmarks replayed collective operations from a trace. More recently, PARAM Full Replay benchmarks (Mystique) [3] successfully used PyTorch ETs (with both compute and collective operations) to generate benchmarks for AI workloads in production. It proposed an automated end-to-end infrastructure to collect and generate benchmarks from the production fleet and demonstrated high accuracy in terms of both performance and system-level metrics. By standardizing the ET schema with Chakra, we can extend this methodology to a wide range of sources; not just ETs collected from PyTorch. We are currently extending the PARAM replay benchmarks to use Chakra ET schema.

6.2 Simulation for Performance Projection

We demonstrate the value of Chakra in simulating the performance of ML training tasks on future systems through two case studies: scaling the number of NPUs and network bandwidth. Our proof-of-concept converts PyTorch ETs to Chakra ETs and uses this to drive an open-source training system simulator (ASTRA-sim).

6.2.1 Implementation. The Chakra infrastructure enables users to utilize any Chakra-compatible simulators for performance modeling. In this paper, we modify ASTRA-sim [9], a distributed ML

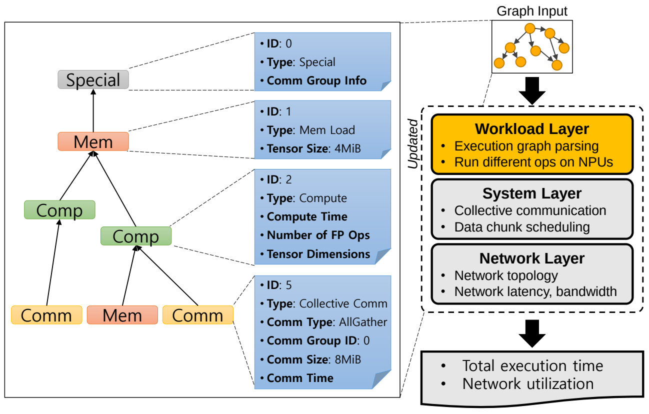


Figure 5: Overview of ASTRA-sim-v2. An execution trace is shown on the left and ASTRA-sim-v2 is shown on the right. Unchanged software layers from ASTRA-sim-v1 are presented in gray, and the updated layer is highlighted with yellow.

system simulator, to make it Chakra-compatible. We choose ASTRA-sim because it models the HW-SW co-design space of training systems, such as workload parallelization, collective communication and scheduling, and networking (Figure 5), and has been validated against real systems.

We denote the original ASTRA-sim as ASTRA-sim-v1 and the updated ASTRA-sim as ASTRA-sim-v2. Figure 5 presents the key changes made to ASTRA-sim-v2 to support ETs. First, we update the workload layer to read `Chakra ET` as its input file and execute the graph, replacing v1's text-based input and manually implemented training loops. Second, we add support for NPUs to execute different operations simultaneously [6], unlike v1 that requires the same operations on each NPU. The graph-input workload layer issues dependency-free nodes from the ET one by one, and an issued node completes after a specific number of cycles. The number of cycles to simulate a compute or communication node can either be read directly from the ET or simulated via ASTRA-sim-v2, which models both NPU's computation capability and network latency/bandwidth. Compute and communication time read directly from the ET effectively models the real system on which the ET was collected, while simulation can be used for modeling future systems[*]. Once a node is completed, dependent nodes become dependency-free if they do not have any other dependencies. The graph-based execution engine repeats this process until all nodes are consumed.

6.2.2 Target Systems. We evaluate two network topologies in this experiment: 2D-torus and DGX2. The 2D-torus is an 8×8 2D-torus topology with 64 NPUs and a link bandwidth of 62GB/s. In contrast, the DGX2 is a hierarchical network topology connected with switches, with network bandwidths for the first and second dimensions set to 600GB/s and 37.5GB/s, respectively.

[*]For instance, a SW company can study the impact of changing the network topology with current NPUs by using the NPU runtime from the ET and simulating the network using an internal Chakra-compatible simulator; A HW vendor can study the impact of their next-generation NPUs while reading communication times from the ET.

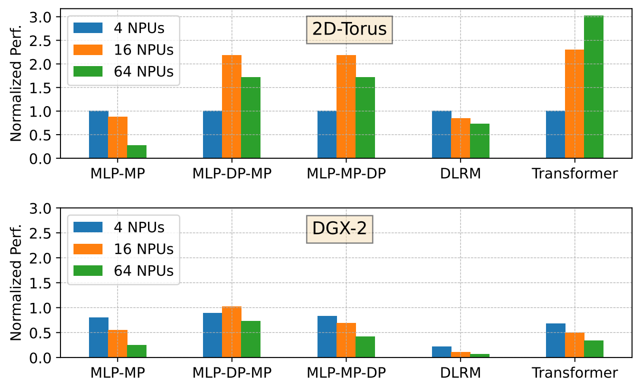
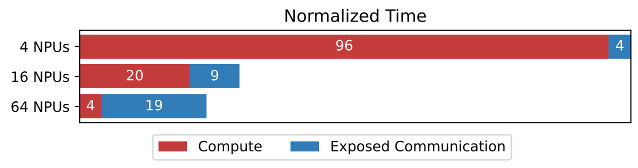
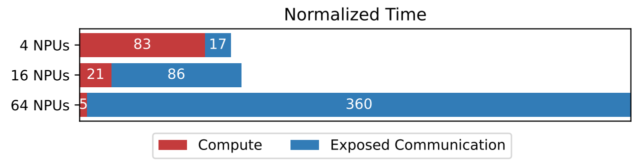


Figure 6: Performance of training tasks while varying the number of NPUs.



(a) `Transformer` on 2D-torus



(b) `MLP-MP` on 2D-torus

Figure 7: Execution time breakdown of training tasks on 2D-torus while varying the number of NPUs.

6.2.3 Target ML Training Tasks. We utilize three models for this study: `MLP`, `DLRM`, and `Transformer`. The `MLP` model is a synthetic workload that consists of six layers. For the `MLP`, we implement model-parallel (MP) and two hybrid parallel schemes: data parallel (DP) along the first network dimension and MP along the second (referred to as `MLP-DP-MP`), and the reverse (`MLP-MP-DP`). The `Transformer` model has 165M parameters and employs hybrid MP-DP using ZeRO-2 optimization for the DP dimension. Meanwhile, the `DLRM` model applies MP for its embedding layers and DP for the MLP layers. The ETs for `MLP` are generated synthetically, while the ETs for `Transformer` and `DLRM` are modeled based on real-world models and converted into the Chakra schema by the converter (Section 5.1).

6.2.4 Experiment Results. ❶ **Scaling the Number of NPUs.** Figure 6 displays the normalized runtime of workloads while varying the number of NPUs between 4, 16, and 64. Performance is defined as the inverse of execution cycles, with values normalized to the 2D-torus topology with four NPUs. Interestingly, performance may not always improve with an increase in the number of NPUs.

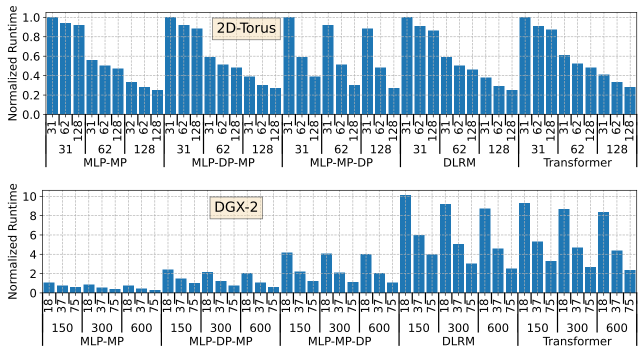


Figure 8: Runtime of training tasks with varying network bandwidth (GBps).

The only case where performance consistently improves is with `Transformer` on the 2D-torus topology. As a compute-dominant model, `Transformer`'s total compute time is 25 times the total communication time on the 2D-torus topology with four NPUs. Therefore, increasing the number of NPUs can considerably enhance `Transformer`'s performance by effectively reducing the compute time. Figure 7 presents the breakdown of normalized execution time for each configuration, with a bar representing the sum of total compute time and exposed communication time, normalized to the 4-NPU configuration. Exposed communication time is defined as the communication time that does not overlap with computation. In `Transformer`, while increasing the number of NPUs leads to increased exposed communication time, the reduction in compute time is significant. However, in `MLP-MP`, exposed communication time dominates and eventually hinders performance when the number of NPUs is increased.

❷ **Varying Network Bandwidth.** In this experiment, we measure the normalized runtime of training loops while varying network bandwidths, with runtime normalized to the 2D-torus network and the lowest network bandwidth combination. The number of NPUs is set to 64. In the 2D-torus topology, we vary the network bandwidth of the first dimension (`dim1`) and the second dimension (`dim2`) between 31, 62, and 128GB/s. In the DGX2-like topology, we vary the dim1 network bandwidth between 150, 300, and 600GB/s, and the dim2 network bandwidth between 18.75, 37.5, and 75GB/s, respectively. Figure 8 illustrates the normalized runtime of workloads while varying the network bandwidth of the first and second dimensions, with dim2 bandwidths displayed on the first row of the xticklabel and dim1 bandwidths on the second row. Generally, training tasks exhibit better performance on the 2D-torus topology compared to the DGX2-like topology. Among the models, `MLP-MP` is the most sensitive training task, with its high sensitivity to network bandwidths stemming from the dominant exposed communication time. On the 2D-torus topology with the lowest bandwidth combination, the exposed communication time of `MLP-MP` is 128 times the total computation time.

6.3 Additional Use Cases

Beside simulations, we can model a ML workload's component or end-to-end performance behavior by combining execution traces with profiling data from real hardware. This is especially useful in large scale ML infrastructures where thousands of models with different complexity and behaviors are being trained daily. GPUs have been widely used as accelerators for large scale ML model training. A number of additional use cases have successfully applied this technology to model GPU performance and solve these problems:

(1) Estimate operator and communication cost for embedding table sharding and load-balancing in distributed training [15, 16].
(2) Iteration latency estimation for performance tuning and speed-of-light modeling [4].

Cost modeling is an important step in the embedding table sharding algorithm. The Dreamshard [16] framework utilizes data collected from traces to guide the computation and communication micro-benchmarks to collect component level performance data. The benchmark results are used to train neural cost models. Millions of different operator configurations are measured through PARAM benchmarks. The trained cost model enabled Dreamshard to outperform the baselines on all the tasks.

To extend beyond component level modeling, the GPU performance model [4] used a critical-path-based algorithm to predict the per-batch training time of ML models by traversing its execution trace. It achieved less than 10% geometric mean average error (GMAE) in all kernel performance modeling, and 4.61% and 7.96% geomean errors for GPU active time and overall E2E training latency prediction including overheads.

The execution trace also provides crucial information for optimization analysis. Operator out-of-order execution via data dependency analysis, combined with data I/O movement and communication modeling, it's now feasible to measure and identify optimization opportunities across thousands of ML workloads at scale.

7 RELATED WORK

The technique of collecting traces to understand system internals and identify bottlenecks is widely adopted and deployed. Google, for instance, collected and released instruction traces from its servers for representative cloud workloads [8]. Chakra is a similar effort aimed at ML systems research, specifically for distributed ML systems. In Chakra, we proposed a common schema for ML execution traces.

As ML tasks are often represented as graphs, numerous graph schemas have been proposed. ONNX is one of the most popular graph schemas, designed to facilitate the exchange of models between different ML frameworks. Although ONNX shares similarities with Chakra, such as enabling data exchange between different teams, the objectives of the two differ. ONNX focuses on the exchange of *models* between various frameworks, while Chakra's primary goal is to exchange *execution traces* between different teams.

When collected in the pre-execution stage, traces closely resemble the model itself, albeit with additional annotations. The traces collected at the pre-execution stage are similar to the graphs presented in prior autotuning studies, which aim to optimize the parallelism strategies of ML models [11–13]. Unity is a representative study in this approach. It optimizes ML models by simultaneously applying algebraic transformations and parallelization. To perform optimizations, Unity defines a new graph schema called the parallel computation graph.

8 CONCLUSION

In this paper, we propose Chakra, an open graph schema for standardizing workload specification capturing key operations and dependencies, aka Execution Trace (ET). In addition, we propose a complementary set of tools/capabilities to enable collection, generation, and adoption of Chakra ETs by a wide range of simulators, emulators, and benchmarks. We believe such a framework would enable wider co-optimization of current AI systems and co-design of future systems for AI inference and training. We solicit feedback from readers and are interested in developing productive collaborations to help build an open industry-wide ecosystem around Chakra schema/tools.

9 ACKNOWLEDGEMENTS

The authors would like to thank Winston Liu, Dan Mihailescu, and Andy Balogh from Ixia/Keysight Technologies for their early feedback and for engaging in insightful discussions that guided the development of the Chakra ET schema and tools. The authors would like to thank Mingyu Liang and Zhongyi Lin for discussions on PARAM replay benchmarking use case.

REFERENCES

[1] Louis Feng. [n. d.]. [PyTorch] Integrate Execution Graph Observer into PyTorch Profiler. https://github.com/pytorch/pytorch/pull/75358.

[2] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks. In *Proceedings of Machine Learning and Systems (MLSys)*.

[3] Mingyu Liang, Wenyin Fu, Louis Feng, Zhongyi Lin, Pavani Panakanti, Shengbao Zheng, Srinivas Sridharan, and Christina Delimitrou. 2023. Mystique: Enabling Accurate and Scalable Generation of Production AI Benchmarks. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*.

[4] Zhongyi Lin, Louis Feng, Ehsan K. Ardestani, Jaewon Lee, John Lundell, Changkyu Kim, Arun Kejariwal, and John D. Owens. 2022. Building a Performance Model for Deep Learning Recommendation Model Training on GPUs. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 227–229. https://doi.org/10.1109/ISPASS55109.2022.00030

[5] Manpreet Singh Minhas. [n. d.]. Computational graphs in PyTorch and TensorFlow. https://towardsdatascience.com/computational-graphs-in-pytorch-and-tensorflow-c25cc40bdcd1.

[6] NVIDIA. [n. d.]. Creating a Communicator. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/communicators.html.

[7] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: Memory optimizations Toward Training Trillion Parameter Models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE.

[8] Parthasarathy Ranganathan and Lee Victor. [n. d.]. Advancing systems research with open-source Google workload traces. https://cloud.google.com/blog/topics/systems/workload-traces-for-google-warehouse-scale-computers.

[9] Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. 2020. ASTRA-SIM: Enabling SW/HW Co-Design Exploration for Distributed DL Training Platforms. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE.

[10] Arun F Rodrigues, K Scott Hemmert, Brian W Barrett, Chad Kersey, Ron Oldfield, Marlo Weston, Rolf Risen, Jeanine Cook, Paul Rosenfeld, Elliot Cooper-Balis, et al. 2011. The structural simulation toolkit. *ACM SIGMETRICS Performance Evaluation Review* 38, 4 (2011), 37–42.

[11] Keshav Santhanam, Siddharth Krishna, Ryota Tomioka, Tim Harris, and Matei Zaharia. 2021. DistIR: An Intermediate Representation and Simulator for Efficient Neural Network Distribution. *arXiv preprint arXiv:2111.05426* (2021).

[12] Michael Schaarschmidt, Dominik Grewe, Dimitrios Vytiniotis, Adam Paszke, Georg Stefan Schmid, Tamara Norman, James Molloy, Jonathan Godwin, Norman Alexander Rink, Vinod Nair, et al. 2021. Automap: Towards Ergonomic Automated Parallelism for ML Models. *arXiv preprint arXiv:2112.02958* (2021).

[13] Fei Wang, Guoyang Chen, Weifeng Zhang, and Tiark Rompf. 2019. Parallel Training via Computation Graph Transformation. In *Proceedings of the International Conference on Big Data (Big Data)*. IEEE, 3430–3439.

[14] William Won, Taekyung Heo, Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. 2023. ASTRA-sim2.0: Modeling Hierarchical Networks and Disaggregated Systems for Large-model Training at Scale. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

[15] Daochen Zha, Louis Feng, Bhargav Bhushanam, Dhruv Choudhary, Jade Nie, Yuandong Tian, Jay Chae, Yinbin Ma, Arun Kejariwal, and Xia Hu. 2022. Autoshard: Automated embedding table sharding for recommender systems. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 4461–4471.

[16] Daochen Zha, Louis Feng, Qiaoyu Tan, Zirui Liu, Kwei-Herng Lai, Bhargav Bhushanam, Yuandong Tian, Arun Kejariwal, and Xia Hu. 2022. DreamShard: Generalizable Embedding Table Placement for Recommender Systems. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 15190–15203. https://proceedings.neurips.cc/paper_files/paper/2022/file/62302a24b04589f9f9cdd5b02c344b6c-Paper-Conference.pdf