https://astra-sim.github.io

https://github.com/mlcommons/chakra

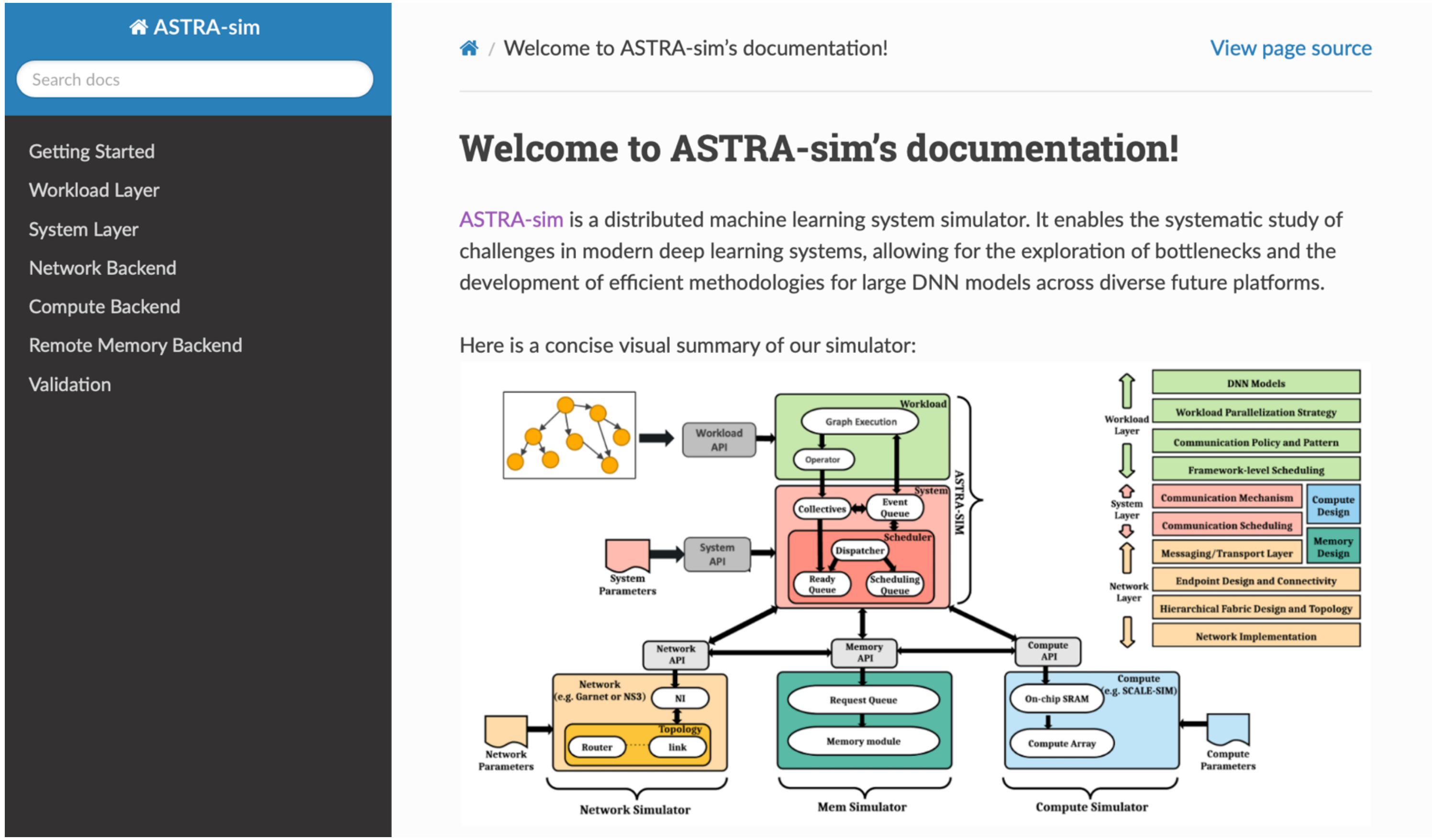
ASTRA-sim Tutorial
MICRO 2024
Nov 3, 2024

ASTRA-sim and Chakra Tutorial:
Wiki and Validation

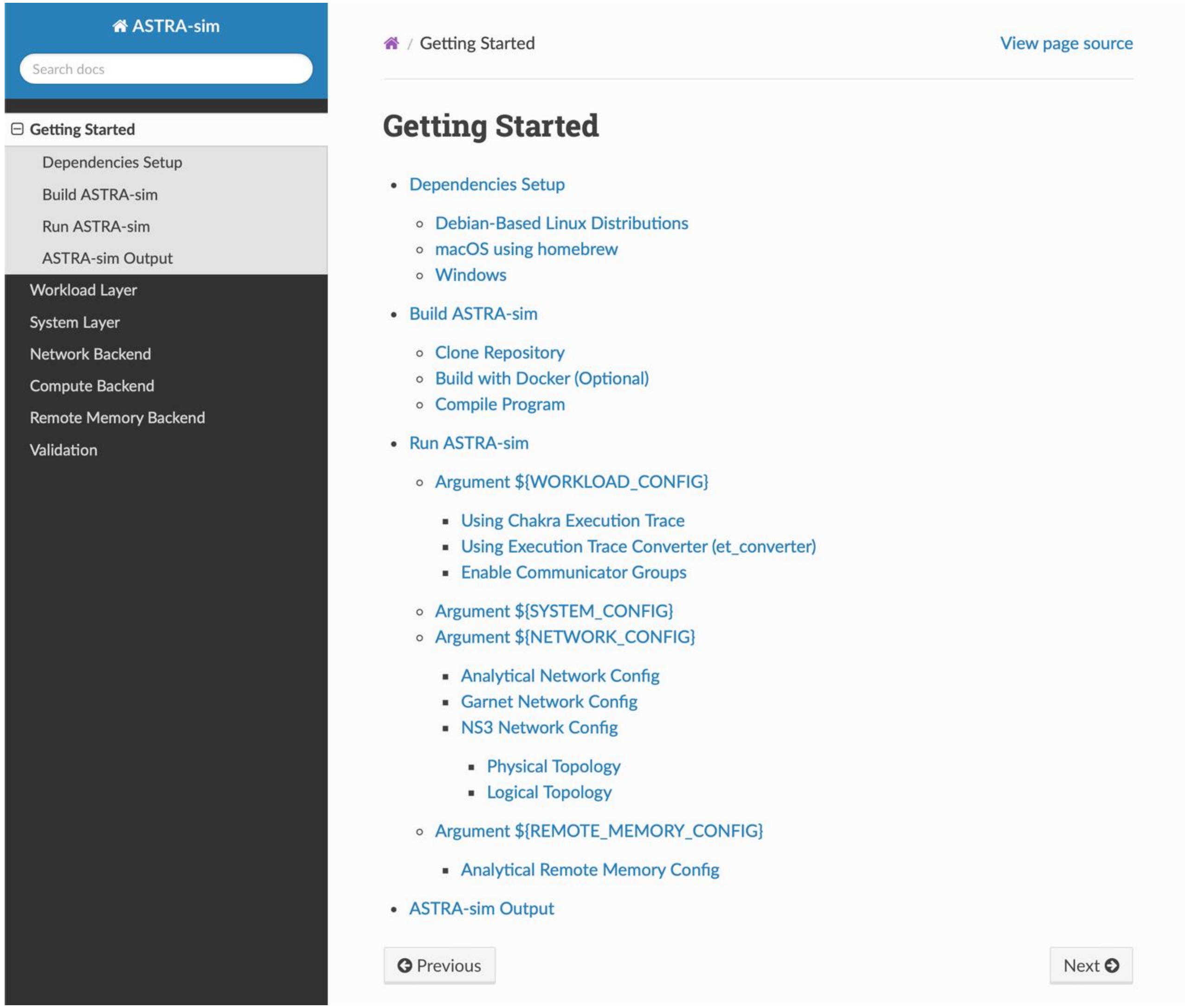
Will Won
Ph.D. Candidate
School of CS, Georgia Institute of Technology
william.won@gatech.edu

ASTRA-sim Wiki Page

- Main Documentation of the ASTRA-sim Framework
  - ***https://astra-sim.github.io/astra-sim-docs/index.html***



ASTRA-sim Wiki Page: Getting Started



ASTRA-sim Wiki Page: Workload

ASTRA-sim

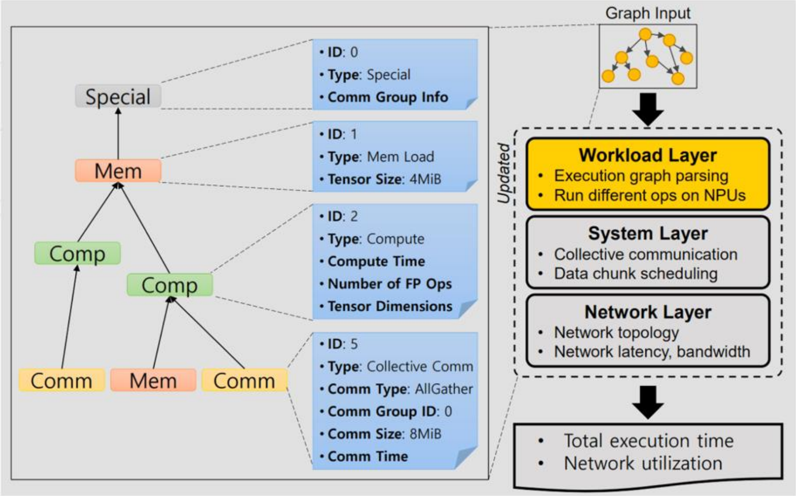
Search docs

- Getting Started
- ⊟ Workload Layer
  - ⊟ Overview
    - ⊞ ASTRA-Sim Workload Layer Overview
- System Layer
- Network Backend
- Compute Backend
- Remote Memory Backend
- Validation

Workload Layer / Overview — View page source

Overview

ASTRA-Sim Workload Layer Overview



The workload layer in ASTRA-Sim plays a pivotal role, enabling users to define and simulate their desired DNN models, parallelization strategies, and training loops efficiently. The transition from ASTRA-Sim 1.0 to 2.0 brought forth significant advancements, enhancing the layer's functionality and operational efficiency.

Evolution from ASTRA-Sim 1.0 to 2.0

Initially, ASTRA-Sim 1.0 laid the groundwork by allowing users to articulate target DNN models, parallelization strategies, and training loops. With the evolution to ASTRA-Sim 2.0, the platform adopted Execution Trace (ET) from Chakra, leveraging a Directed Acyclic Graph (DAG) format for more streamlined and structured processing.

ASTRA-sim Wiki Page: System

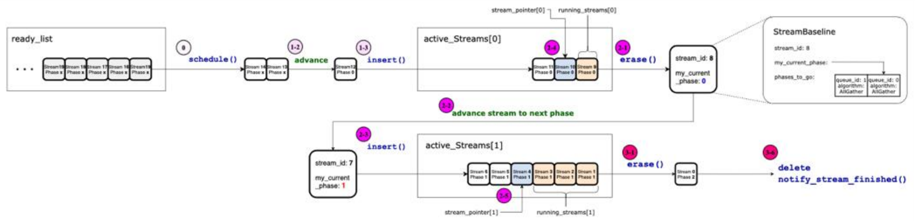
ASTRA-sim

Search docs

- Getting Started
- Workload Layer
- ⊟ System Layer
  - **Collective Scheduler**
  - Collective Implementation
  - Input Files for Collective API
- Network Backend
- Compute Backend
- Remote Memory Backend
- Validation

System Layer / Collective Scheduler   View page source

Collective Scheduler



The system layer has a collective scheduler which schedules and dispatches collectives. Even if the dependencies for multiple non-dependent collectives have been resolved in the workload layer (such as in a Data Parallel case), a single NPU cannot issue dozens of collectives at once, due to hardware limits. Core to the scheduler is a set of queues called `active_Streams`.

Each queue holds `StreamBaseline` objects, which are depicted at the top right corner of the image. A `StreamBaseline` object represents a stream (i.e. collective), which consists of multiple collective phases. The variable `phases_to_go` is a queue holding these phases. The pointer `my_current_phase` points to the phase currently being executed.

For each stream, the function `proceed_to_next_vnet_baseline` is critical in advancing the collective phases and moving the stream object between one queue to another. This function is called in the following possible cases:

1. When a stream has been removed from `ready_list` and is about to be inserted into `active_Streams` for the first time.
2. When a stream has finished one phase and is ready to wait for the next phase.
3. When a stream has finished its last phase.

ASTRA-sim Wiki Page: Network

Analytical Network
latest
Search docs

- Installation
  - Install Dependencies
  - Clone Repository
  - Compile Analytical Network Simulator
  - Run Simulation
- Input Format
- Frequently Asked Questions

Installation   View page source

Installation

This page explains how to compile an analytical network simulator as a standalone binary.

Tip

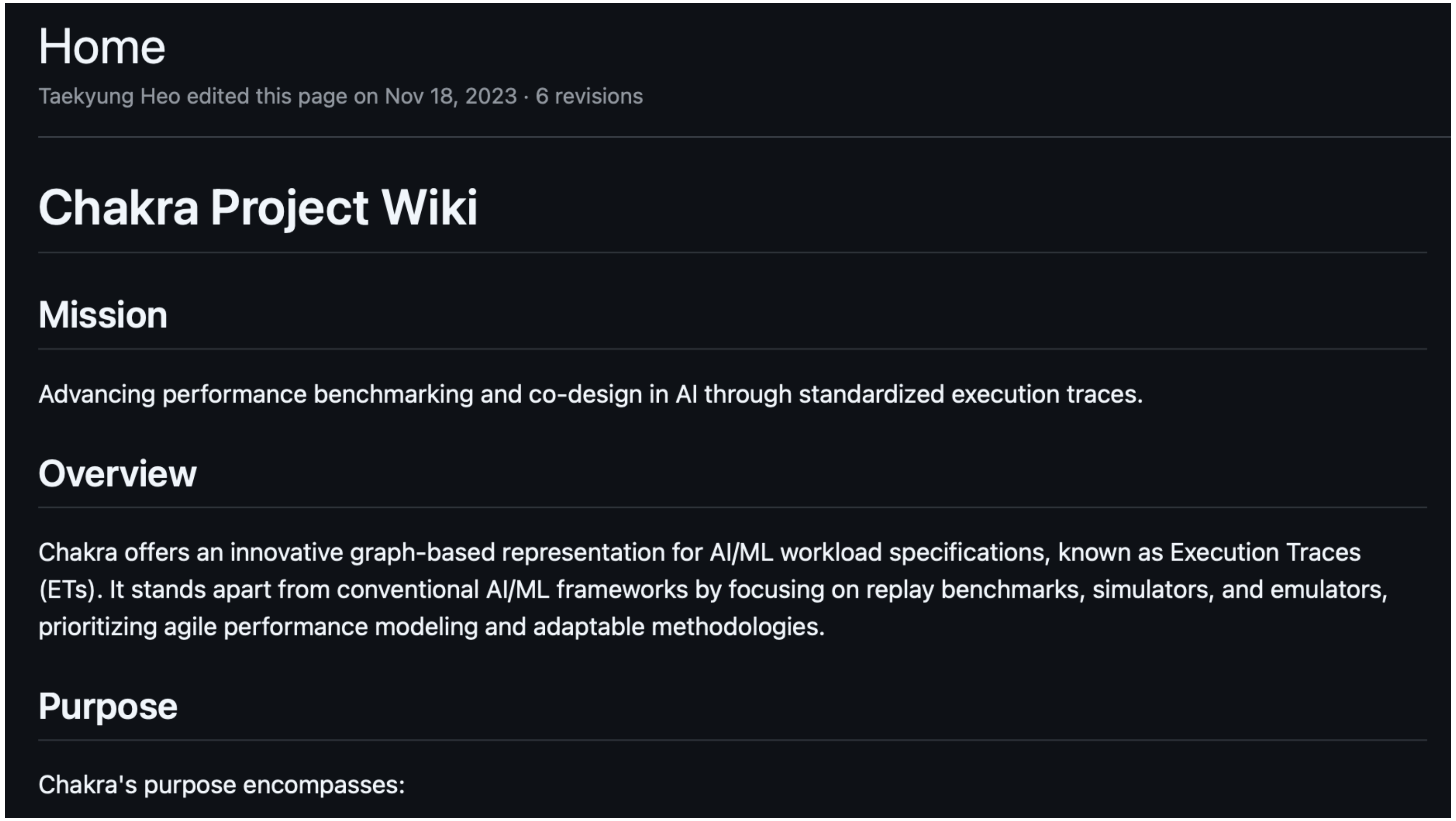
This page explains how to use the analytical network simulator as a standalone program. If you want to use the analytical network simulator as a backend to the ASTRA-sim, please refer to this link.

- Install Dependencies
  - macOS
  - Debian-based Linux
- Clone Repository
- Compile Analytical Network Simulator
  - Overview
  - Additional Options
    - Build in Debug Mode
    - Compilation Target
- Run Simulation
  - Congestion_Unaware Simulation
  - Congestion_Aware Simulation

Previous   Next

Chakra Wiki Page

- Main documentation of the MLCommons/Chakra Project
  - ***https://github.com/mlcommons/chakra/wiki***



Home

Taekyung Heo edited this page on Nov 18, 2023 · 6 revisions

Chakra Project Wiki

Mission

Advancing performance benchmarking and co-design in AI through standardized execution traces.

Overview

Chakra offers an innovative graph-based representation for AI/ML workload specifications, known as Execution Traces (ETs). It stands apart from conventional AI/ML frameworks by focusing on replay benchmarks, simulators, and emulators, prioritizing agile performance modeling and adaptable methodologies.

Purpose

Chakra's purpose encompasses:

Chakra Wiki Page: Installation

Installation Guide

Joongun Park edited this page on Sep 18 · 4 revisions

Chakra User Guide

Installation

Step 1: Set up a Virtual Environment

It's advisable to create a virtual environment using Python 3.10.2.

```
# Create a virtual environment
$ python3 -m venv chakra_env

# Activate the virtual environment
$ source chakra_env/bin/activate
```

Step 2: Install Chakra

With the virtual environment activated, install the Chakra package using pip.

```
# Install package from source
$ pip install .

# Install latest from GitHub
$ pip install https://github.com/mlcommons/chakra/archive/refs/heads/main.zip

# Install specific revision from GitHub
$ pip install https://github.com/mlcommons/chakra/archive/ae7c671db702eb1384015bb2618dc753eed787f2.zip
```

Execution Trace Visualizer (chakra_visualizer)

This tool visualizes execution traces in various formats. Here is an example command:

```
$ chakra_visualizer \
    --input_filename /path/to/chakra_et
    --output_filename /path/to/output.[graphml|pdf|dot]
```

Execution Trace Jsonizer (chakra_jsonizer)

Provides a readable JSON format of execution traces:

```
$ chakra_jsonizer \
    --input_filename /path/to/chakra_et \
    --output_filename /path/to/output_json
```

Timeline Visualizer (chakra_timeline_visualizer)

Visualizes the execution timeline of traces. This tool serves as a reference implementation for visualizing the simulation of Chakra traces. After simulating Chakra traces, you can visualize the timeline of operator executions. Update the simulator to present when operators are issued and completed. Below is the format needed:

```
issue,<dummy_str>=npu_id,<dummy_str>=curr_cycle,<dummy_str>=node_id,<dummy_str>=node_name
callback,<dummy_str>=npu_id,<dummy_str>=curr_cycle,<dummy_str>=node_id,<dummy_str>=node_name
...
```

Chakra Wiki Page: Trace Collection

Chakra Execution Trace Collection - A Comprehensive Guide on Merging PyTorch and Kineto Traces

Joongun Park edited this page on Sep 24 · 30 revisions

Authors: Saeed Rashidi, Joongun Park, Abhilash Kolluri, and Taekyung Heo

1. Introduction

This document outlines the process of collecting and simulating Chakra execution traces for performance projection and design space exploration using a simulator. This document covers the collection of PyTorch execution traces (ET) and Kineto traces, their linker, and the subsequent conversion into Chakra execution traces, a standardized format that encapsulates both CPU and GPU operation information.

2. Overview of Trace Collection and Simulation Methodology

Chakra execution traces and the related toolchains enable the simulation of execution traces on a simulator. The figure below illustrates how the end-to-end flow works. The process begins by collecting traces from a PyTorch model. There are two types of traces collected from PyTorch: PyTorch ET and Kineto trace. We need to collect two different types of traces because each trace type covers aspects that the other cannot. While PyTorch ETs focus on CPU operators with explicit dependencies between them, Kineto traces encode GPU operators with their start and end times. To understand the differences between further, please refer to the table below, which highlights their differences and roles. After collecting these traces, we use a merger tool (`chakra_trace_link`) to merge them into a single execution trace, known as PyTorch ET+. This format essentially follows the PyTorch ET schema but also encodes GPU operators and their dependencies. Subsequently, these traces are converted into the Chakra schema using the converter (`chakra_converter`). Finally, you can use any Chakra-compatible simulator, with ASTRA-sim currently serving as a reference implementation.

Pages

- Home
  - Ove
- Commu
  - Wo
  - Dis
  - Gith
- Chakra
  - Cha
  - v0.0
- Tools &
  - Rep
  - Sim
- Getting
  - Inst
  - Cha

Validation

- Validation effort is actively underway
  - ***https://astra-sim.github.io/astra-sim-docs/validation/validation.html***

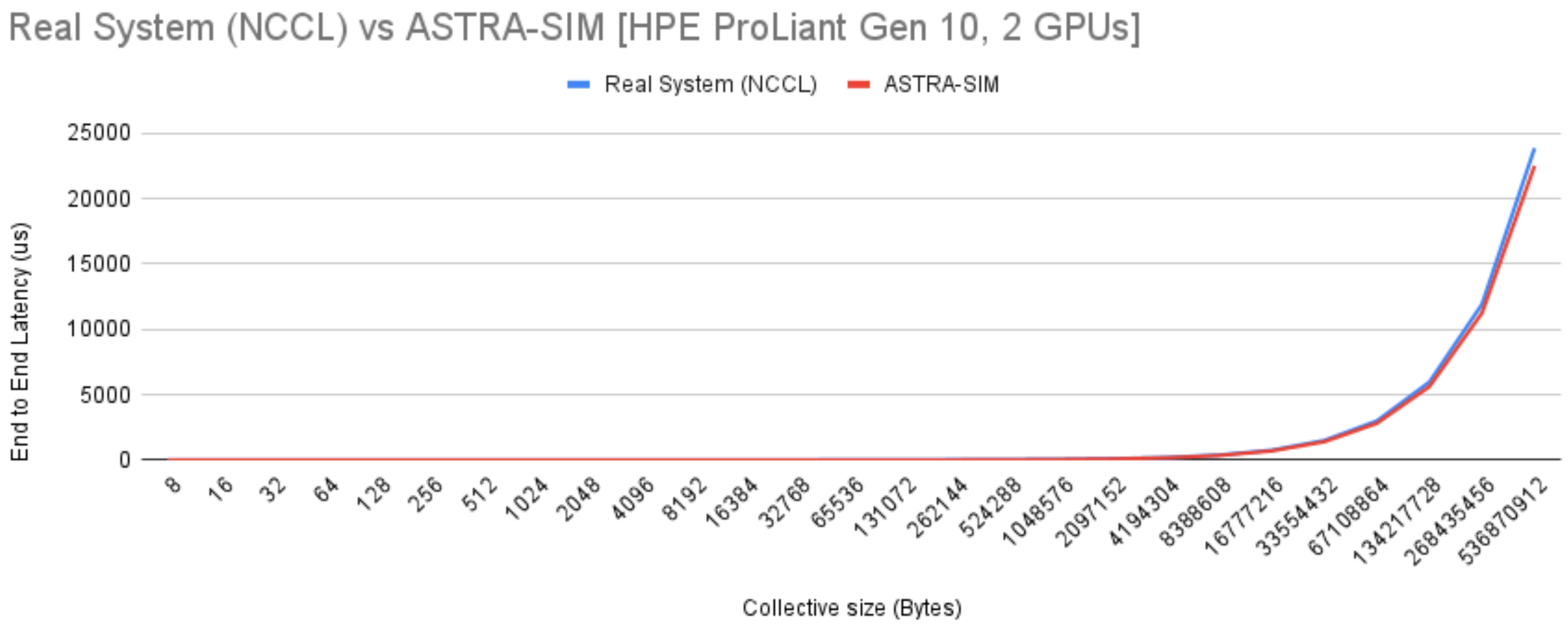
- Currently validated at the **collective communication level**
  - In progress: **end-to-end workload-level** validation

Validation Process

- Run the real system measurement
  - Measure: Practical **network bandwidth** and **endpoint delay**
- Run the ASTRA-sim simulation with the measured value
- Compare the result

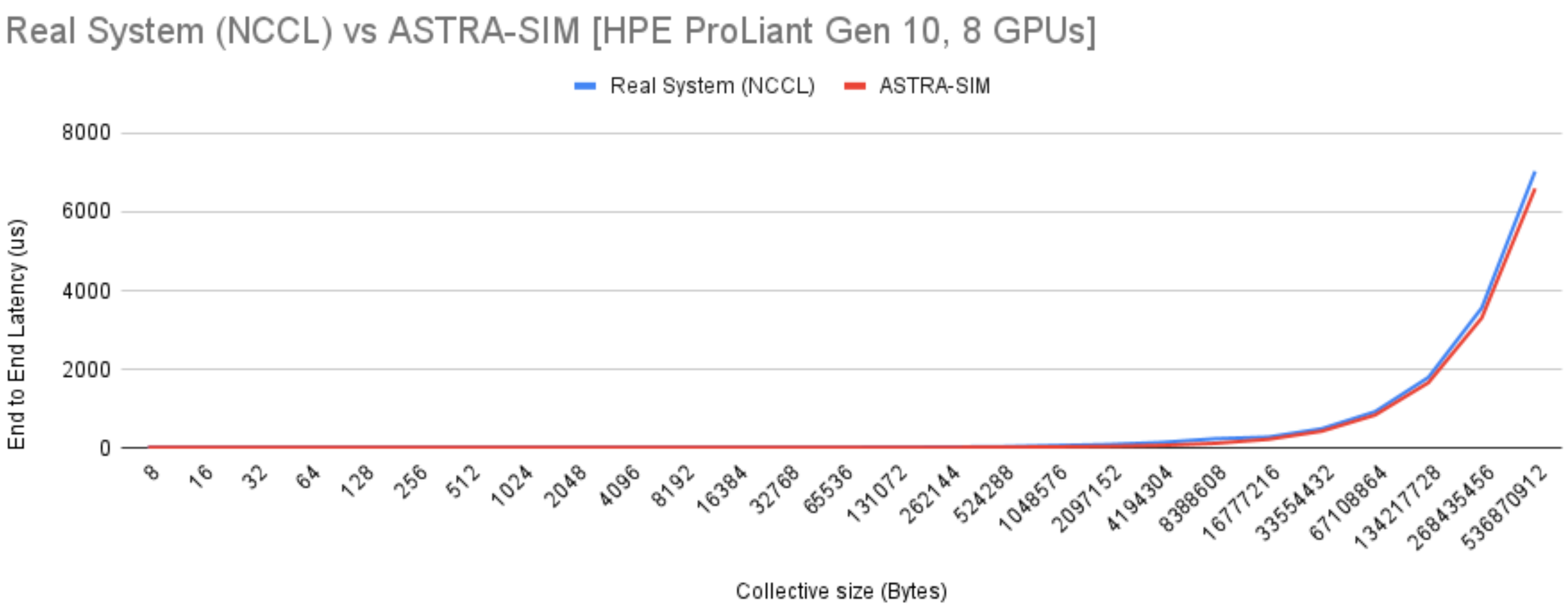
Validation Example: 2-GPU HPE Cluster

- Simplest validation setup



geomean error: 11.4%

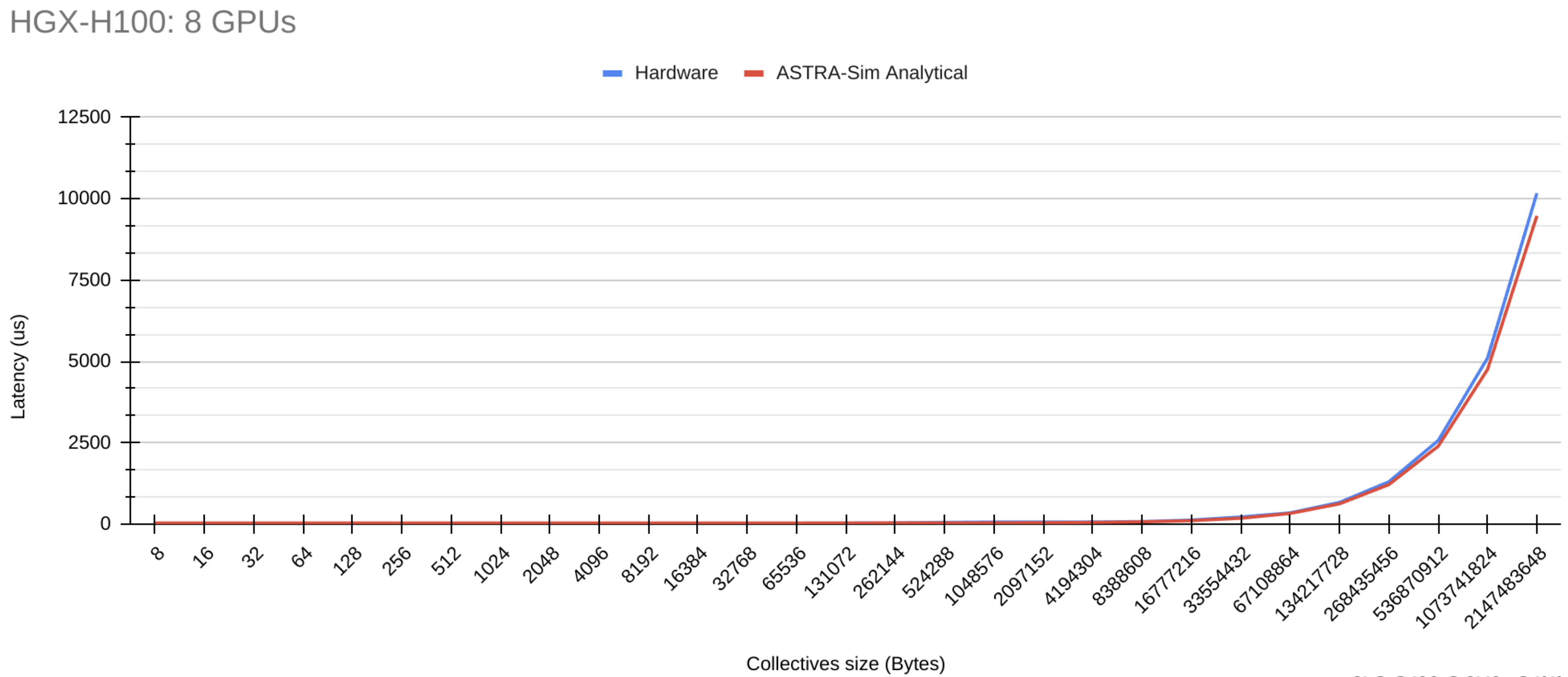
Validation Example: 8-GPU HPE Cluster



geomean error: 2.8%

Validation Example: H100 System

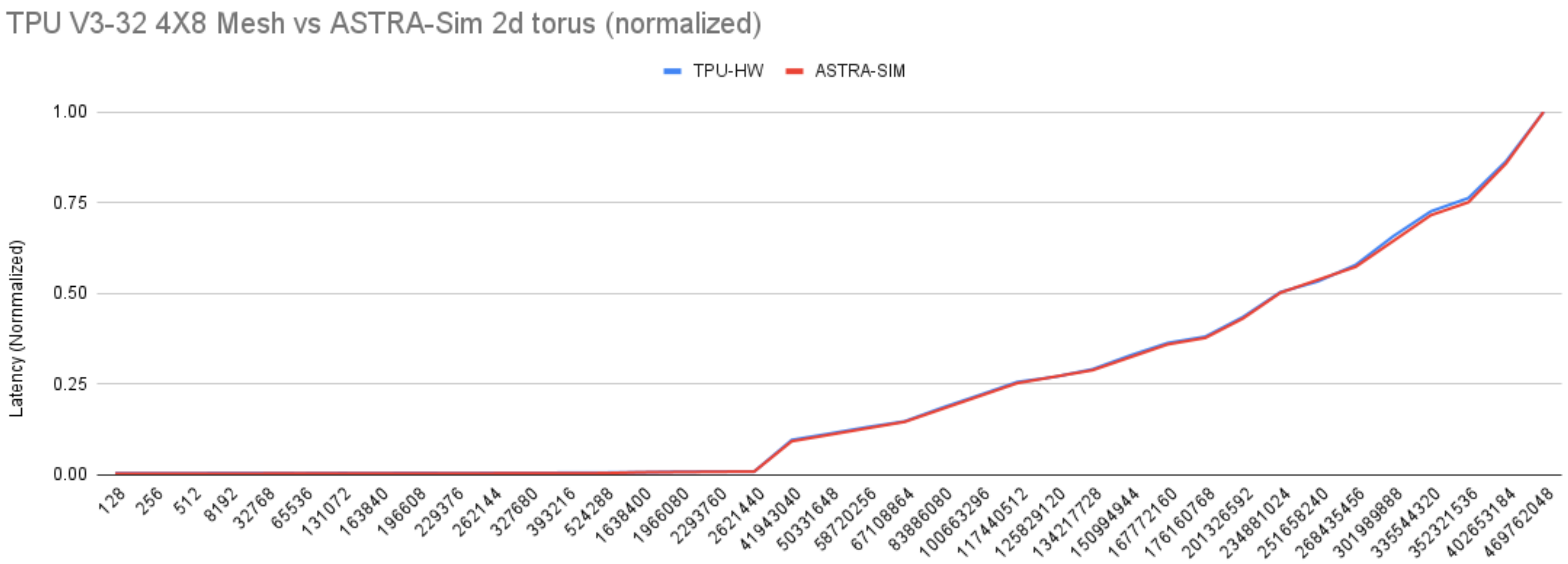
- Measured BW: 741.34 GB/s (82.37%)
  - Nominal BW: 900 GB/s



geomean error: 9.69%

Validation Example: TPUv3 Cluster

- 32-TPU 2D Torus System



Work In Progress

- **Wiki** is continuously being upgraded
  - e.g., adding documentations for the API specs
  - adding more validation results

- Validation and **real system modeling**
  - Plan: measure practical efficiency and endpoint delay
  - and implement a flag to turn this efficiency/delay on
  - e.g., ***./astra-sim –system=H100***