



Chakra: AI Workload Twin for Benchmarking and Co-design

Taekyung Heo
Software Engineer @ NVIDIA

AI SW/HW Co-design Requirements

- 1 **Replay:** Reproduce AI workload on actual hardware
- 2 **Simulation:** Capture exact behavior for future system co-design
- 3 **Analysis:** Identify performance bottlenecks and opportunity
- 4 **Sharing:** Obfuscate IP-sensitive details of customer models

Chakra Execution Trace

Graph is the Representation!

- Extensible and standardized graph format to represent AI workloads
 - Nodes: primitive operators and tensor objects with attributes and timing
 - Edges: data and control dependency
- Benefits
 - Isolate comms and compute operators
 - Graph transformations to obscure sensitive IP
 - Operator, dependencies, and timing for replay, simulation, and analysis
 - Flexible to represent both workloads and collective implementations



ML Commons

07.31.2023 - San Francisco, CA

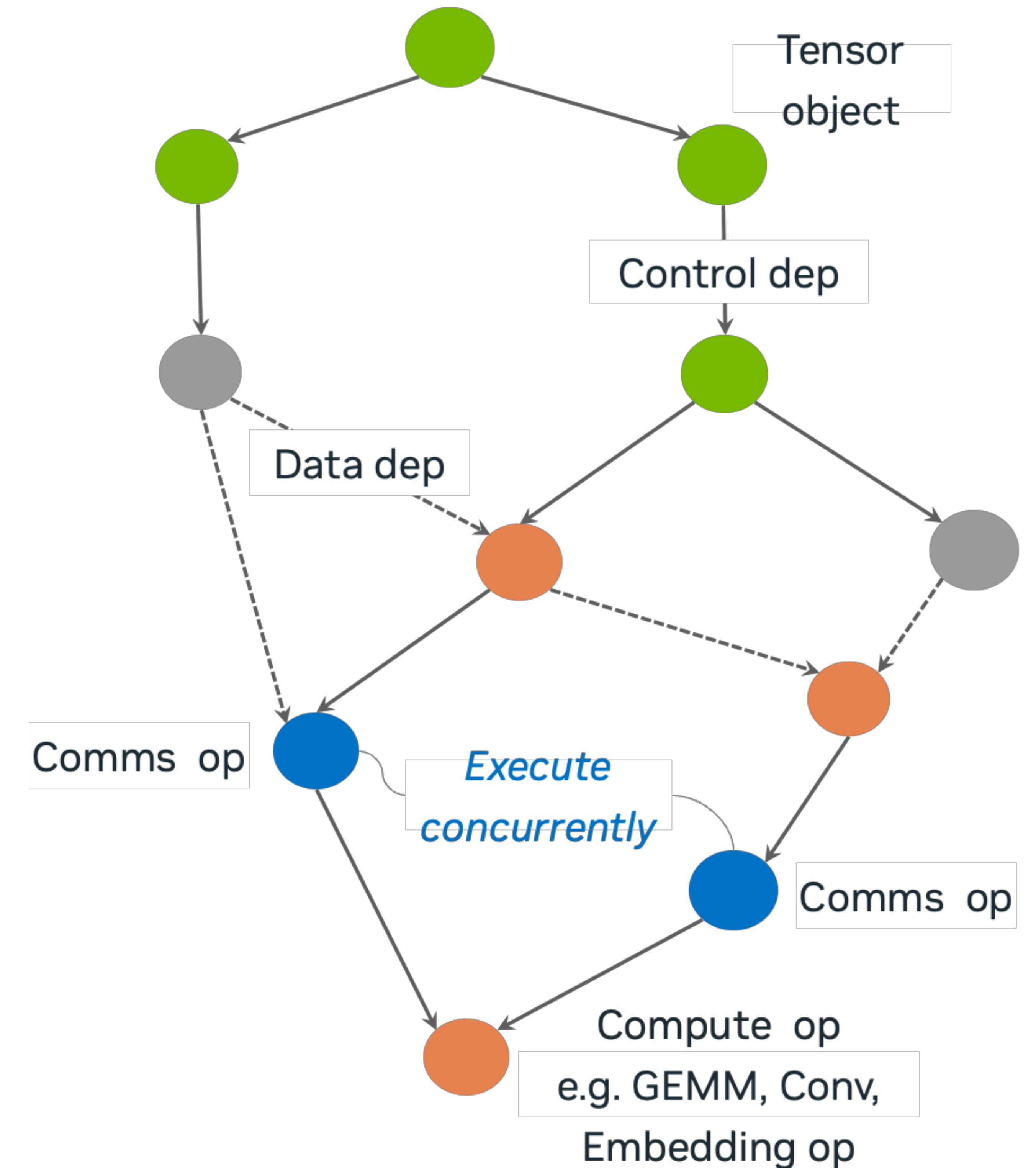
Chakra: Advancing Benchmarking and Co-design for Future AI Systems

Announcing Chakra, execution traces and benchmarks working group

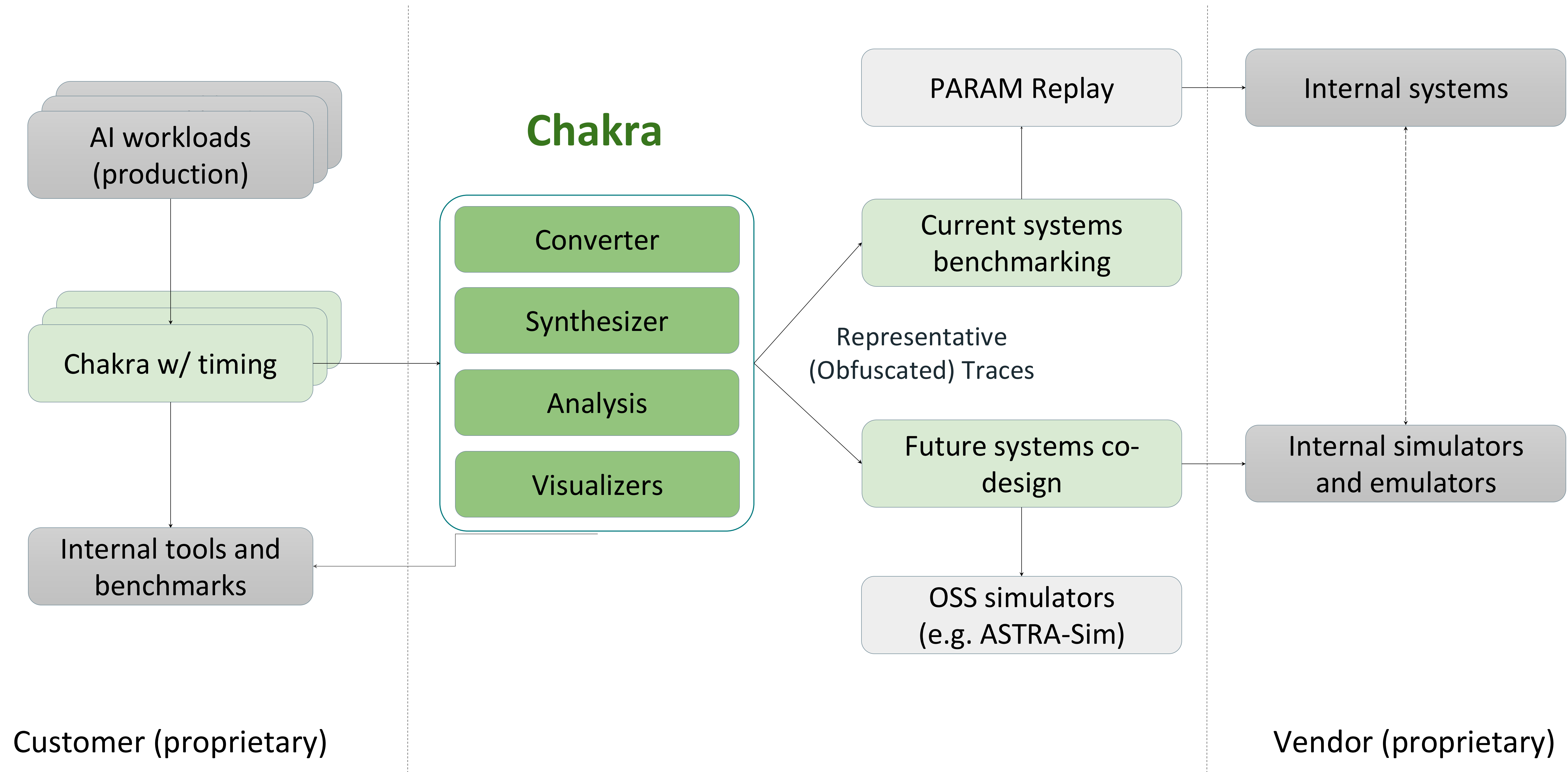
Menu ☰



MLCommons



Chakra Ecosystem



Chakra Execution Trace Schema

https://github.com/mlcommons/chakra/blob/main/schema/protobuf/et_def.proto



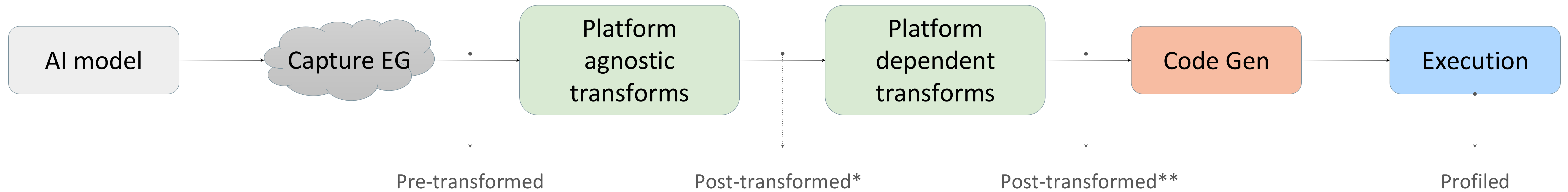
mlcommons / chakra

```
132  message Node {
133      uint64 id = 1;
134      string name = 2;
135      NodeType type = 3;
136
137      // Control and data dependencies
138      repeated uint64 ctrl_deps = 4;
139      repeated uint64 data_deps = 5;
140
141      // Timing information
142      uint64 start_time_micros = 6;
143      uint64 duration_micros = 7;
144
145      IOInfo inputs = 8;
146      IOInfo outputs = 9;
147      repeated AttributeProto attr = 10;
148  }
```

```
108  enum NodeType {
109      INVALID_NODE = 0;
110      METADATA_NODE = 1;
111      MEM_LOAD_NODE = 2;
112      MEM_STORE_NODE = 3;
113      COMP_NODE = 4;
114      COMM_SEND_NODE = 5;
115      COMM_RECV_NODE = 6;
116      COMM_COLL_NODE = 7;
117  }
118
119  enum CollectiveCommType {
120      ALL_REDUCE = 0;
121      REDUCE = 1;
122      ALL_GATHER = 2;
123      GATHER = 3;
124      SCATTER = 4;
125      BROADCAST = 5;
126      ALL_TO_ALL = 6;
127      REDUCE_SCATTER = 7;
128      REDUCE_SCATTER_BLOCK = 8;
129      BARRIER = 9;
130  }
```

```
5  message AttributeProto {
6      string name = 1;
7      string doc_string = 2;
8
9      oneof value {
10         double double_val = 3;
11         DoubleList double_list = 4;
12         float float_val = 5;
13         FloatList float_list = 6;
14         int32 int32_val = 7;
15         Int32List int32_list = 8;
16         int64 int64_val = 9;
17         Int64List int64_list = 10;
18         uint32 uint32_val = 11;
19         Uint32List uint32_list = 12;
20         uint64 uint64_val = 13;
21         Uint64List uint64_list = 14;
22         sint32 sint32_val = 15;
23         Sint32List sint32_list = 16;
24         sint64 sint64_val = 17;
25         Sint64List sint64_list = 18;
26         fixed32 fixed32_val = 19;
27         Fixed32List fixed32_list = 20;
28         fixed64 fixed64_val = 21;
29         Fixed64List fixed64_list = 22;
30         sfixed32 sfixed32_val = 23;
31         Sfixed32List sfixed32_list = 24;
32         sfixed64 sfixed64_val = 25;
```

Chakra Traces: Source and Intent



- Pre-transformed: original model
- Post-transformed: optimized graph (may or may not be platform dependent)
- Profiled: graph executed on a specific platform
 - Chakra ET today for PT eager mode

Chakra Execution Trace Collection

Chakra Host Execution Trace (PyTorch Execution Trace)

```
from torch.profiler import _ExperimentalConfig,  
ExecutionTraceObserver  
  
et = ExecutionTraceObserver()  
et.register_callback("pytorch_et.json")  
et.start()  
...  
et.stop()  
et.unregister_callback()
```

Chakra Device Execution Trace (Kineto Trace)

```
import torch  
  
def trace_handler(prof):  
    prof.export_chrome_trace("./kineto_trace.json")  
  
def main():  
    with torch.profiler.profile(  
        activities=[  
            torch.profiler.ProfilerActivity.CPU,  
            torch.profiler.ProfilerActivity.CUDA,  
        ],  
        schedule=torch.profiler.schedule(  
            wait=0,  
            warmup=10,  
            active=1),  
        record_shapes=True,  
        on_trace_ready=trace_handler,  
    ) as prof:  
        ...  
        prof.step()
```

Chakra Execution Trace Collection

<https://github.com/mlcommons/chakra/wiki/Chakra-Execution-Trace-Collection-%E2%80%90-A-Comprehensive-Guide-on-Merging-PyTorch-and-Kineto-Traces>

The screenshot shows a GitHub Wiki page for the repository 'mlcommons / chakra'. The page title is 'Chakra Execution Trace Collection - A Comprehensive Guide on Merging PyTorch and Kineto Traces', edited by Joongun Park on Sep 24 with 30 revisions. The page content includes an introduction and an overview of trace collection and simulation methodology. A sidebar on the right contains a table of contents with 16 pages.

mlcommons / chakra

Code Issues 21 Pull requests 9 Actions Projects Wiki Security 1 Insights

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 16

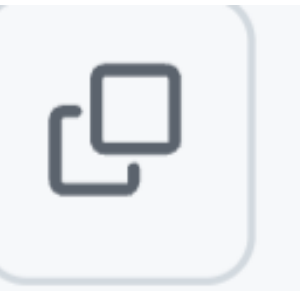
- Home
 - [Overview](#)
- Community
 - [Working Group](#)
 - [Discord](#)
 - [GitHub Repository](#)
- Chakra Schema Release Notes
 - [Chakra Schema](#)
 - [v0.0.4](#)
- Tools & Applications
 - [Replay Tools](#)
 - [Simulation Tools](#)
- Getting Started
 - [Installation Guide](#)
 - [Chakra Execution Trace Collection](#)
 - [Running Simulation with Chakra](#)



Chakra Execution Trace Collection

Chakra Host Execution Trace

```
{
  "schema": "1.0.1", "pid": 2839879, "time": "2024-01-11 21:10:53", "start_ts": 1325266906,
  "nodes": [
    {
      "name": "[pytorch|profiler|execution_trace|thread]", "id": 2, "rf_id": 0, "parent": 1, "fw_parent": 0,
      "inputs": [], "input_shapes": [], "input_types": [],
      "outputs": [], "output_shapes": [], "output_types": []
    },
    {
      "name": "aten::lift_fresh", "id": 5, "rf_id": 1, "parent": 2, "fw_parent": 0, "seq_id": 0, "scope": 0,
      "inputs": [[3,4,0,1048576,8,"cpu"]], "input_shapes": [[1024,1024]], "input_types": ["Tensor(double)"],
      "outputs": [[3,4,0,1048576,8,"cpu"]], "output_shapes": [[1024,1024]], "output_types": ["Tensor(double)"]
    },
    {
      "name": "aten::empty_strided", "id": 8, "rf_id": 4, "parent": 7, "fw_parent": 0, "seq_id": -1, "scope": 0,
      "inputs": [[1024,1024],[1024,1],6,0,"cpu",false], "input_shapes": [[[]],[[]],[[]],[[]],[[]],[[]],[[]],[[]]], "input_types": ["Tensor(float)"],
      "outputs": [[9,10,0,1048576,4,"cpu"]], "output_shapes": [[1024,1024]], "output_types": ["Tensor(float)"]
    }
  ]
}
```



Chakra Execution Trace Collection

Chakra Host Execution Trace

Global Metadata

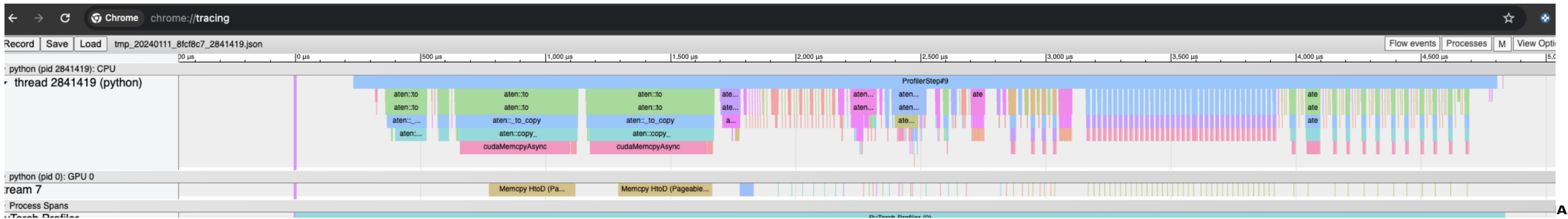
```
{  
  "schema": "1.0.2-chakra.0.0.4", "pid": 836680, "time": "2023-10-22 19:26:48", "start_ts": 927411725,  
  "nodes": [  
    {  
      Per-node Info  
      "id": 2, "name": "[pytorch|profiler|execution_trace|thread]", "ctrl_deps": 1,  
      "inputs": {"values": [], "shapes": [], "types": []},  
      "outputs": {"values": [], "shapes": [], "types": []},  
      "attributes": [{"name": "rf_id", "type": "uint64", "value": 0}, {"name": "fw_parent", "type": "uint64", "value": 0},  
{"name": "seq_id", "type": "int64", "value": -1}, {"name": "scope", "type": "uint64", "value": 7}, {"name": "tid", "type":  
"uint64", "value": 1}, {"name": "fw_tid", "type": "uint64", "value": 0}, {"name": "op_schema", "type": "string", "value": ""}]  
    },  
    ...  
  ]  
}
```

Per-node Info

Chakra Execution Trace Collection

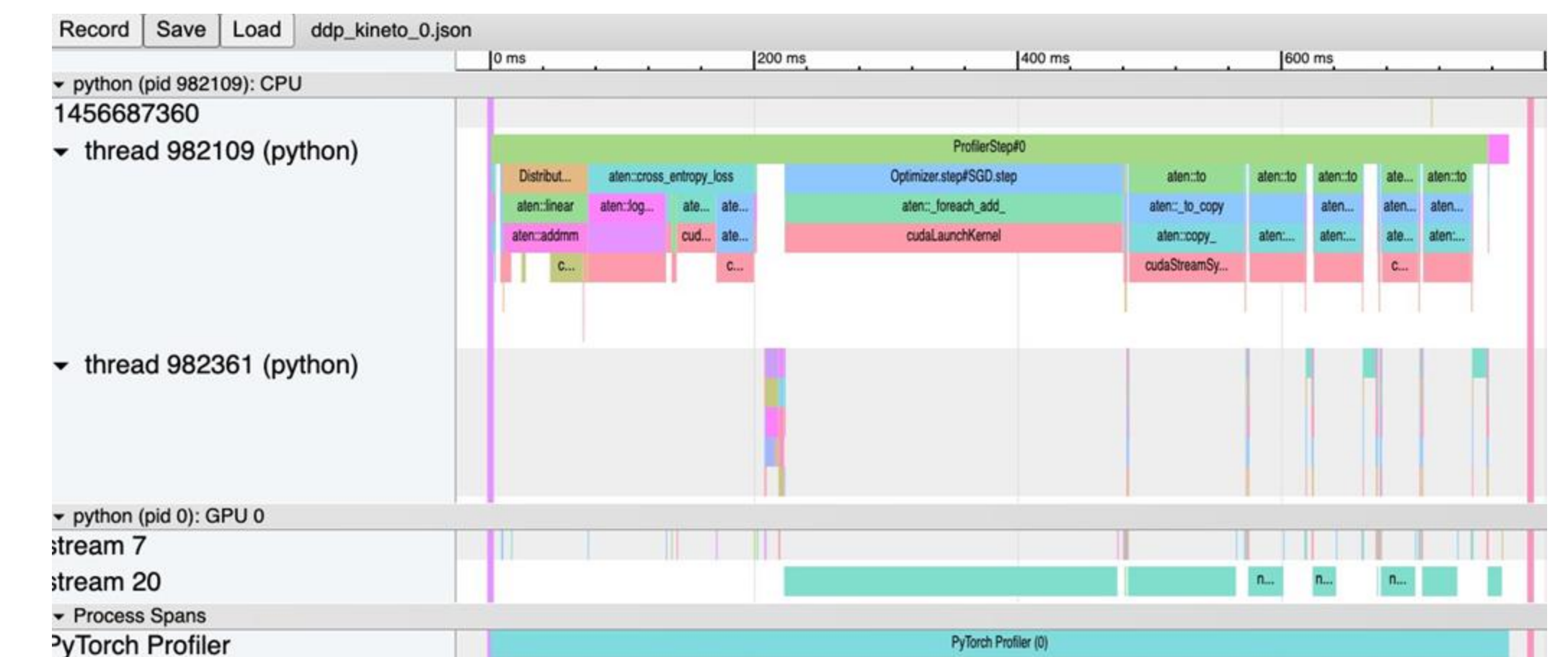
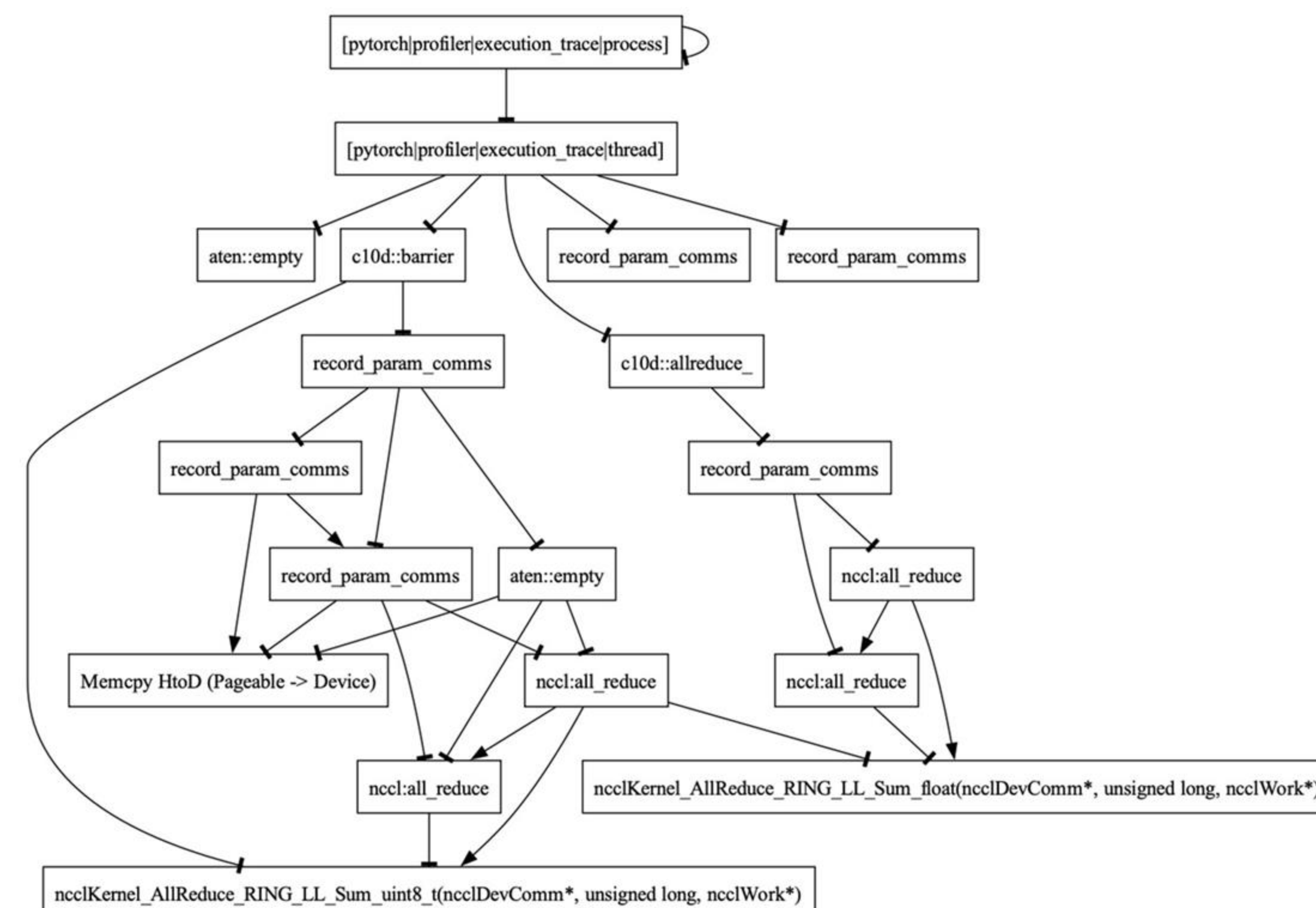
Chakra Device Execution Trace

```
{
  "schemaVersion": 1,
  "deviceProperties": [
    {
      "id": 0, "name": "NVIDIA H100u80GBnHBM3", "totalGlobalMem": 84943110144,
      "computeMajor": 9, "computeMinor": 0,
      "maxThreadsPerBlock": 1024, "maxThreadsPerMultiprocessor": 2048,
      "regsPerBlock": 65536, "regsPerMultiprocessor": 65536, "warpSize": 32,
      "sharedMemPerBlock": 49152, "sharedMemPerMultiprocessor": 23347,
      "numSms": 132, "sharedMemPerBlockOptin": 232448
    },
    {
      "id": 1, "name": "NVIDIA H100u80GBnHBM3", "totalGlobalMem": 84943110144,
      "computeMajor": 9, "computeMinor": 0,
      "maxThreadsPerBlock": 1024, "maxThreadsPerMultiprocessor": 2048,
      "regsPerBlock": 65536, "regsPerMultiprocessor": 65536, "warpSize": 32,
      "sharedMemPerBlock": 49152, "sharedMemPerMultiprocessor": 23347,
      "numSms": 132, "sharedMemPerBlockOptin": 232448
    }
  ],
}
```



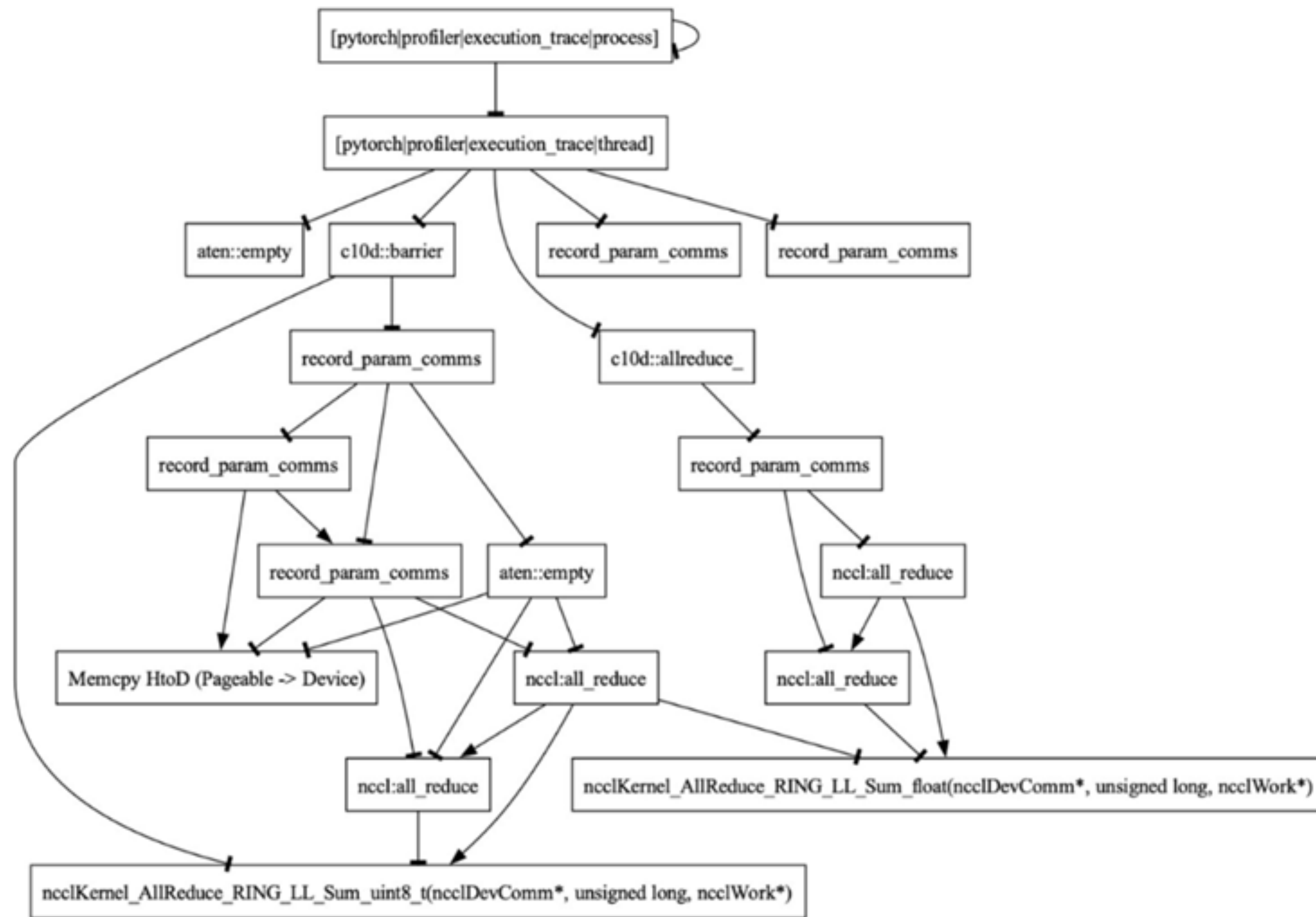
Chakra Execution Trace Types

	Chakra Host Execution Trace (PyTorch Execution Trace)	Chakra Device Execution Trace (Kineto Trace)
Encoded Operators	CPU	CPU & GPU
Encoded Dependencies	Control dependencies Data dependencies	No explicit dependencies
Encoded Metadata	Input / output values, shapes, types	Duration, GPU kernel information
Missing Metadata	Duration, GPU kernel information	Input / output values, shapes, types

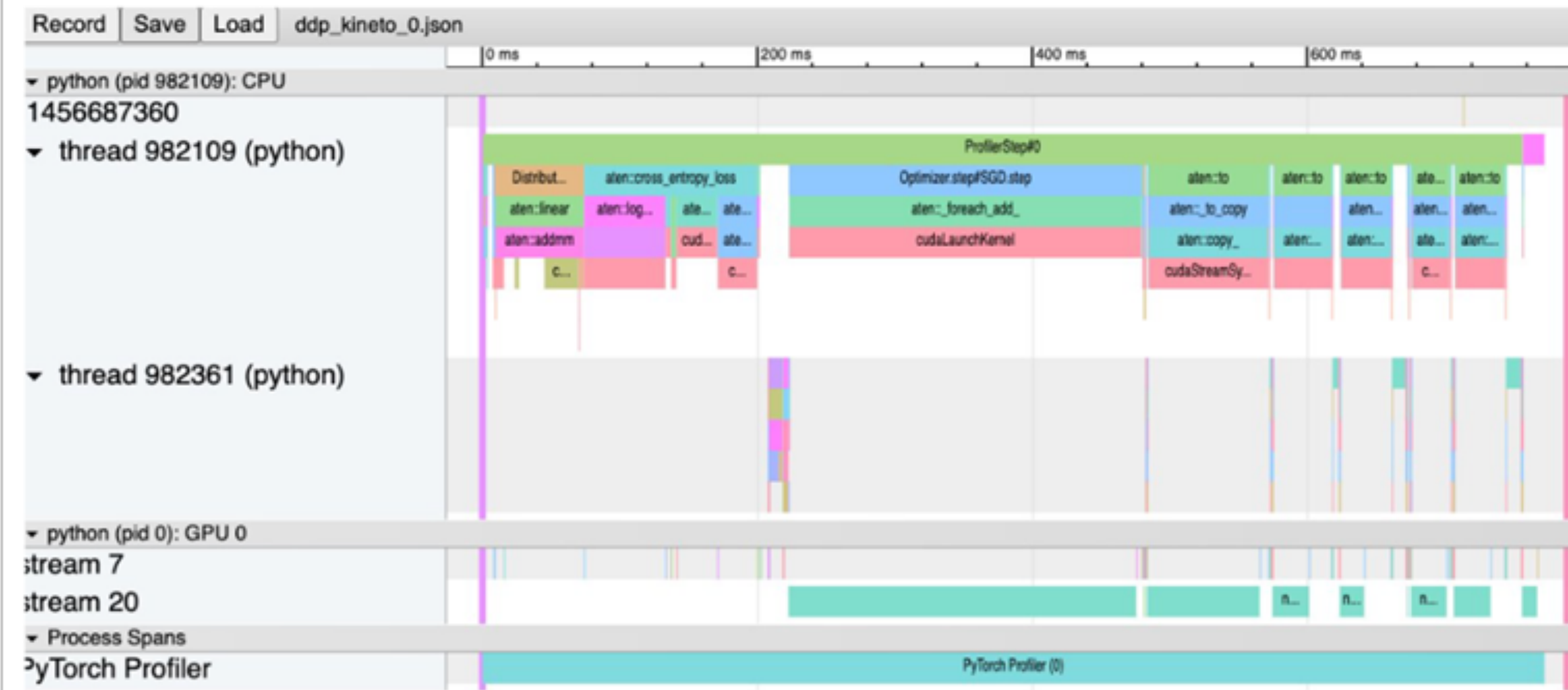


Chakra Execution Trace Types

Chakra Host Execution Trace (PyTorch Execution Trace)



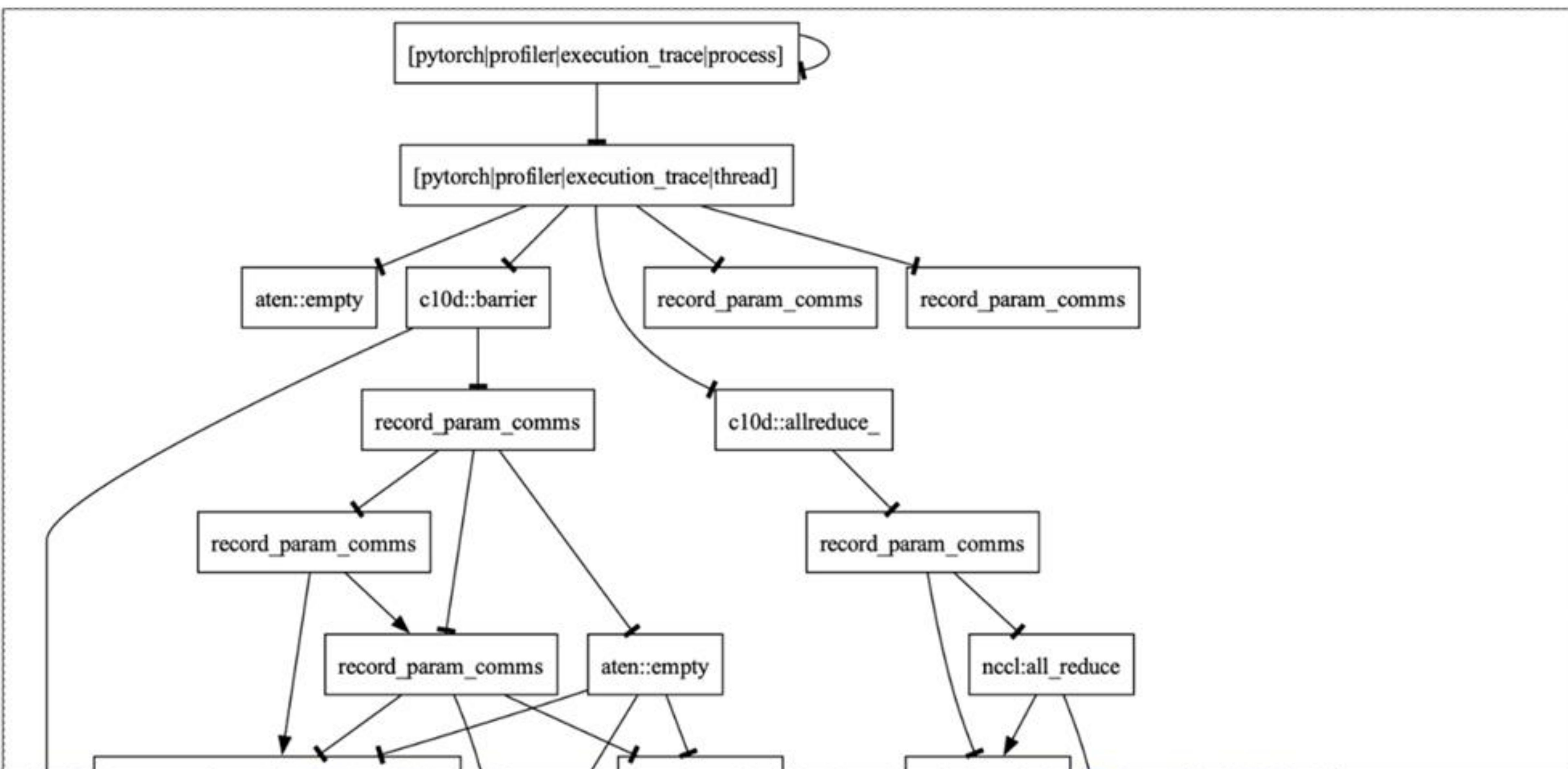
Chakra Device Execution Trace (Kineto Trace)



Chakra Execution Trace Types

Understanding Different Types of Dependencies

- Control Dependency
- Data Dependency



```
def pytorch|profiler|execution_thread|thread():
    aten::empty()
    c10d::barrier()
    c10d::allreduce_
    record_param_comms()
    record_param_comms()
```

```
def c10d::barrier():
    record_param_comms()
```

```
def record_param_comms():
    record_param_comms()
    record_param_comms()
    aten::empty()
```

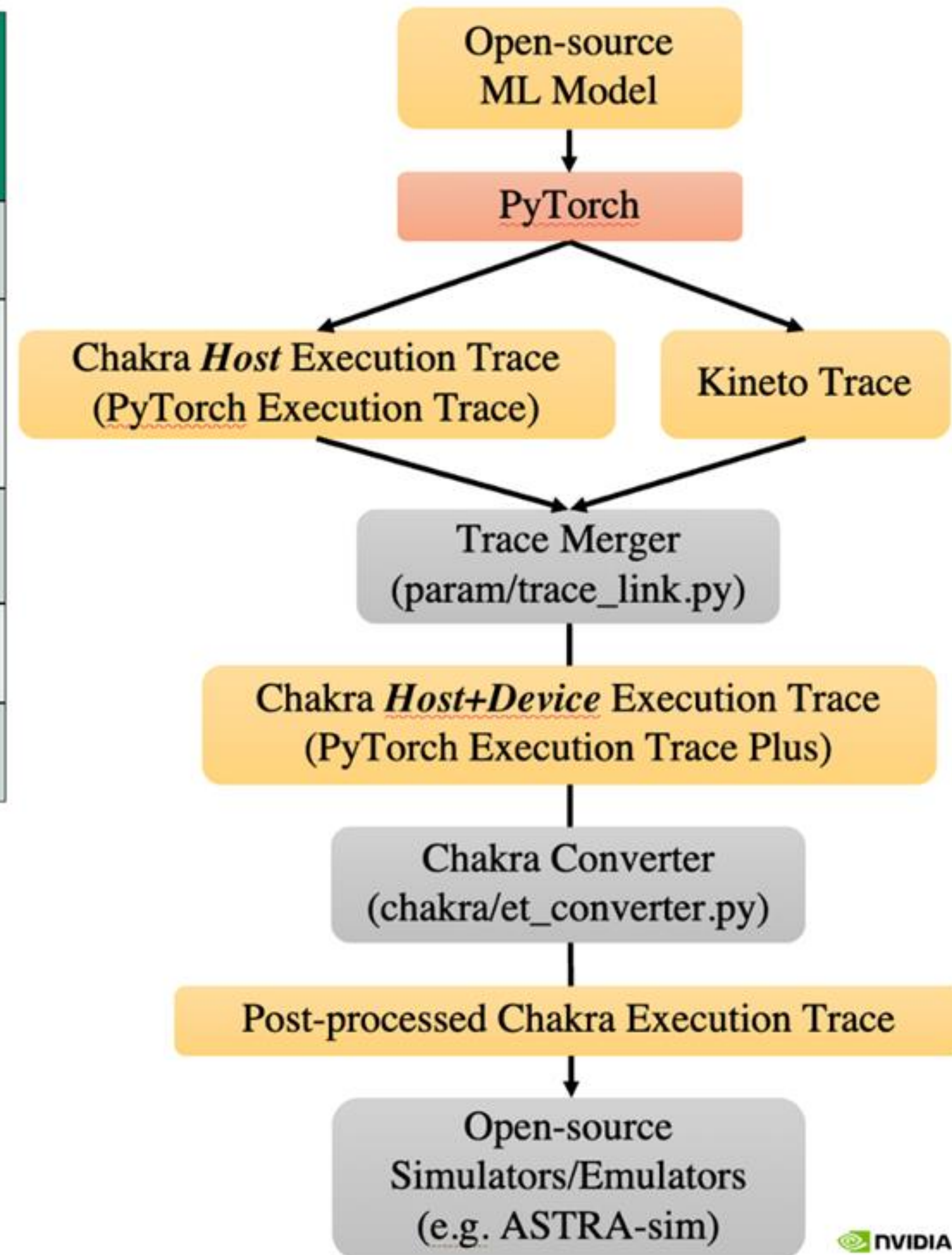
```
"nodes": [
{
  "name": "[pytorch|profiler|execution_trace|thread]", "id": 2, "rf_id": 0, "parent": 1, "fw_parent":
  "inputs": [], "input_shapes": [], "input_types": [],
  "outputs": [], "output_shapes": [], "output_types": []
},
{
  "name": "aten::empty", "id": 3, "rf_id": 1, "parent": 2, "fw_parent": 0, "seq_id": -1, "scope": 0,
  "inputs": [[1], 0, "<None>", "cuda", "<None>", "<None>"], "input_shapes": [[[]], [], [], [], [], []], "input_
  "outputs": [[4, 5, 0, 1, 1, "cuda:0"]], "output_shapes": [[1]], "output_types": ["Tensor(unsigned char)"]
},

```

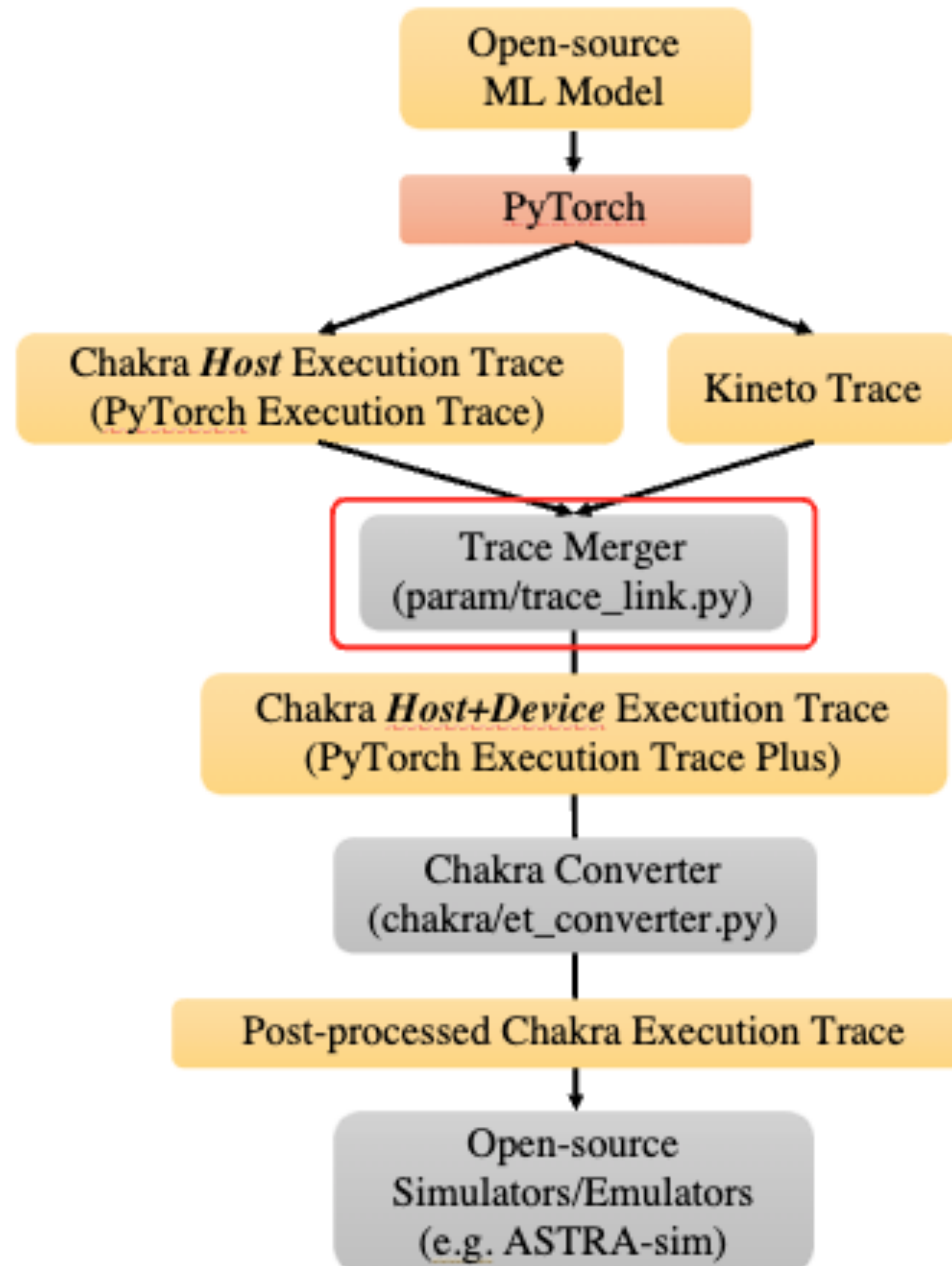
Chakra Execution Trace Types

	Chakra Host Execution Trace (PyTorch Execution Trace)	Chakra Device Execution Trace (Kineto Trace)	Chakra Host+Device ET	Post-processed Chakra ET
Encoded Operators	CPU	CPU & GPU	CPU & GPU	CPU & GPU
Encoded Dependencies	Control dependency Data dependency	No explicit dependencies	Control dependency Data dependency	Control dependency Data dependency Simulation dependency
Input/Output Values, Shapes, Types	✓	✗	✓	✓
Duration	✓	✓	✓	✓
GPU Kernel	✗	✓	✓	✓

- Trace Merger
- Link PyTorch ops with Kineto ops
 - Encode durations
 - Encode additional metadata
 - Add GPU operators

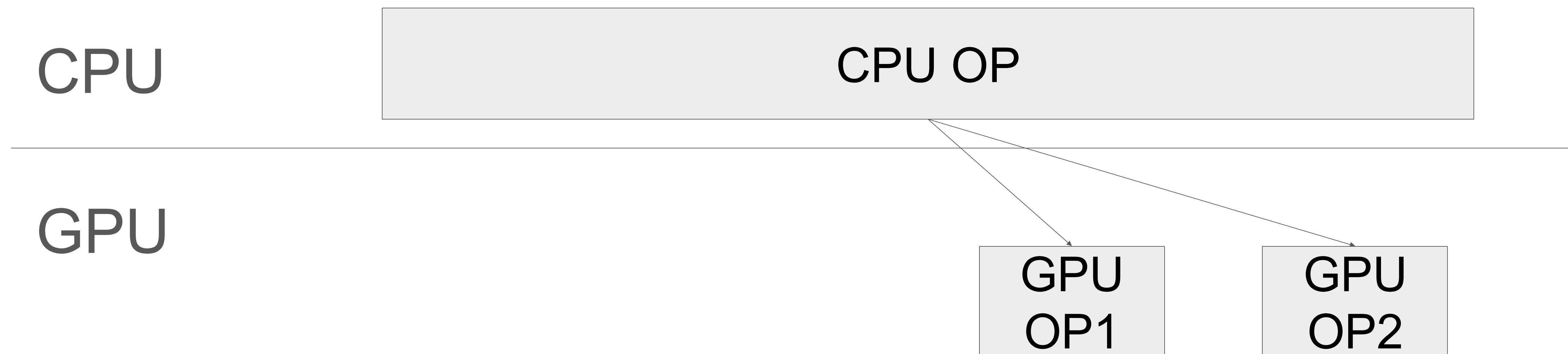


Chakra Execution Trace Postprocessing Steps



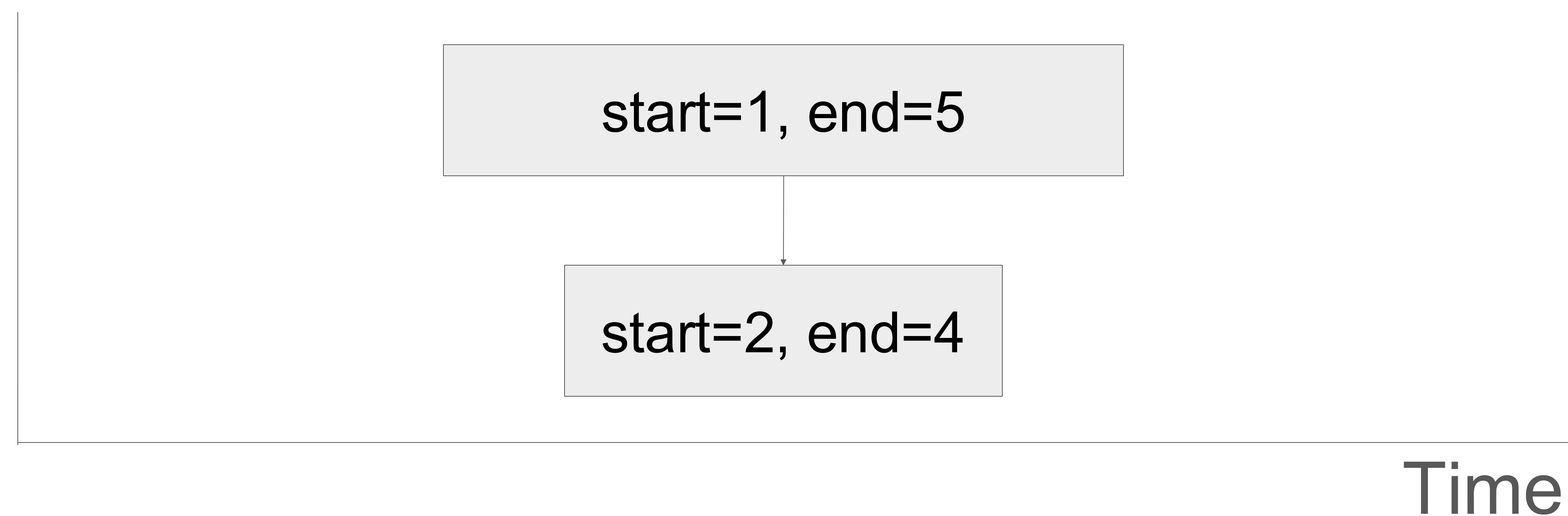
Trace Merger Internals

- Determine dependencies considering
 - Kineto arrow (CPU ops to GPU ops)



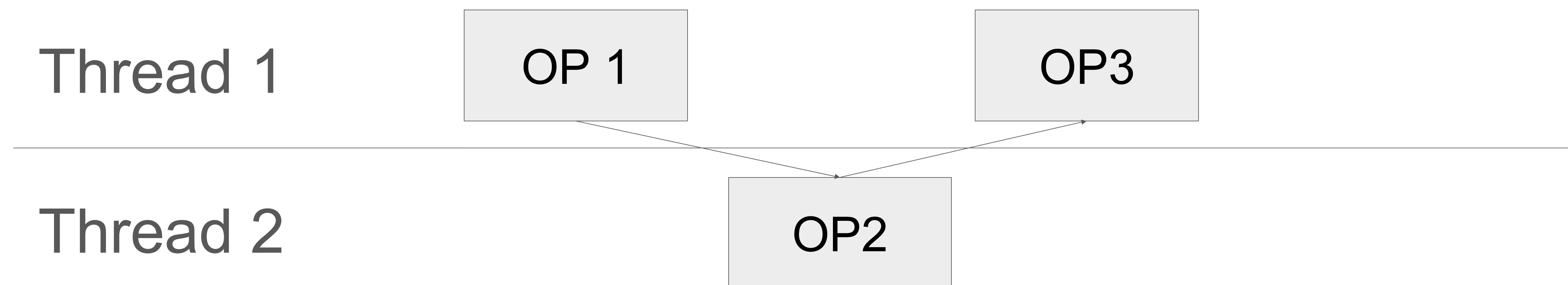
Trace Merger Internals

- Determine dependencies considering
 - Kineto arrow (CPU ops to GPU ops)
 - **Time window analysis**



Trace Merger Internals

- Determine dependencies considering
 - Kineto arrow (CPU ops to GPU ops)
 - Time window analysis
 - **Inter-thread dependency**



Trace Merger Internals

- Determine dependencies considering
 - Kineto arrow (CPU ops to GPU ops)
 - Time window analysis
 - Inter-thread dependency
 - **Intra-stream dependency**

GPU Stream N

OP1

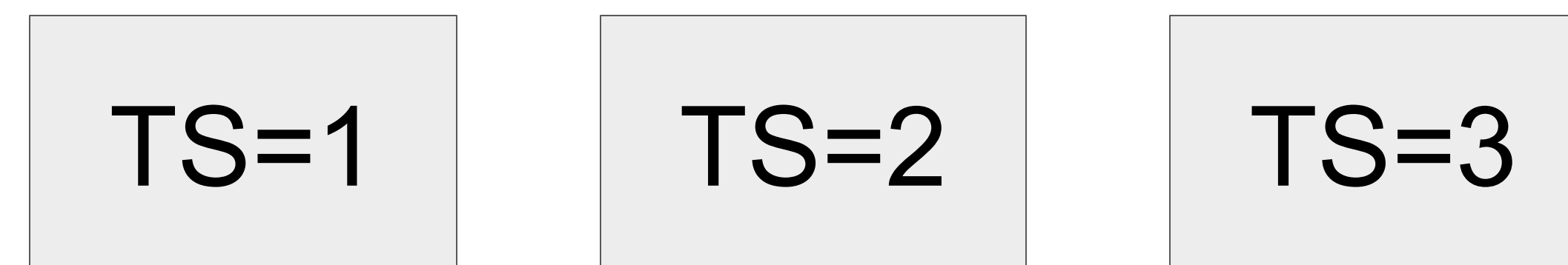
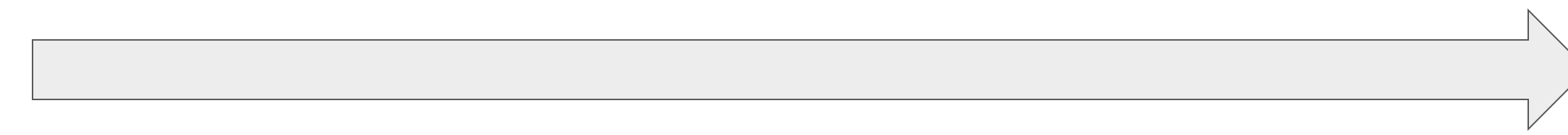
OP2

OP3

⋮

Trace Merger Internals

Sort with **time stamp** (Kineto)

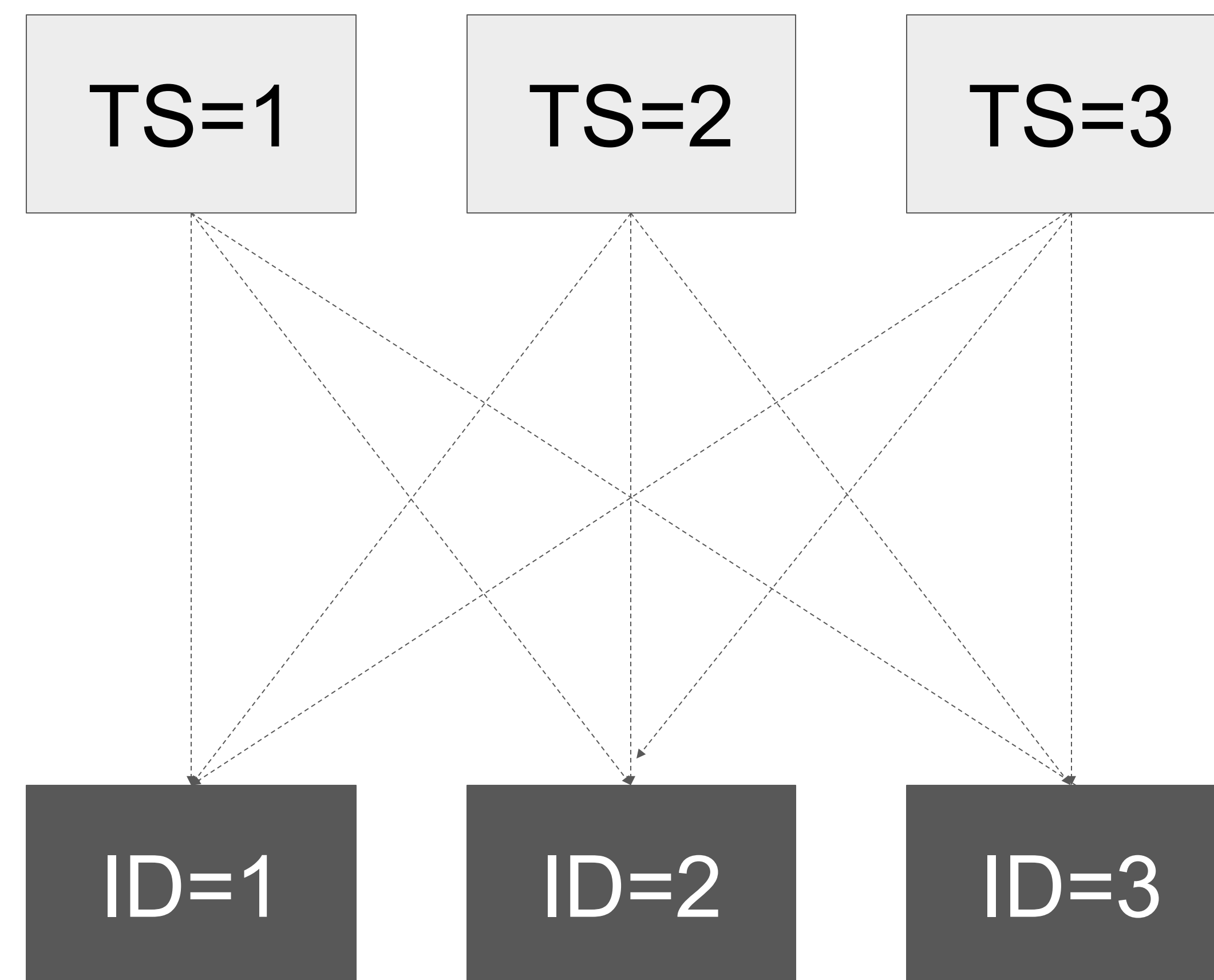


Sort with **Ops ID** (Execution Trace Observer)



Trace Merger Internals

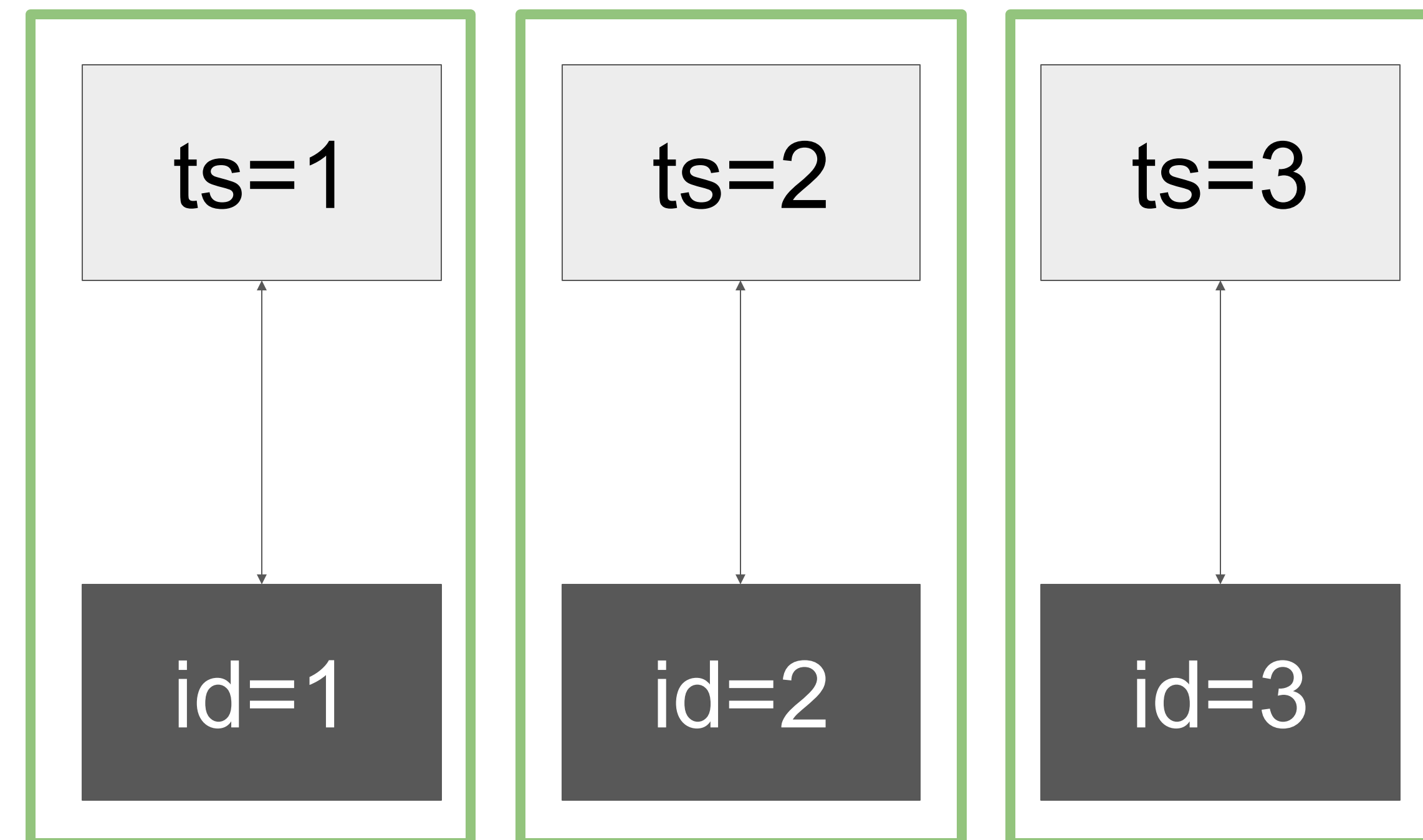
Match with **operation name**



-----> Pattern matching

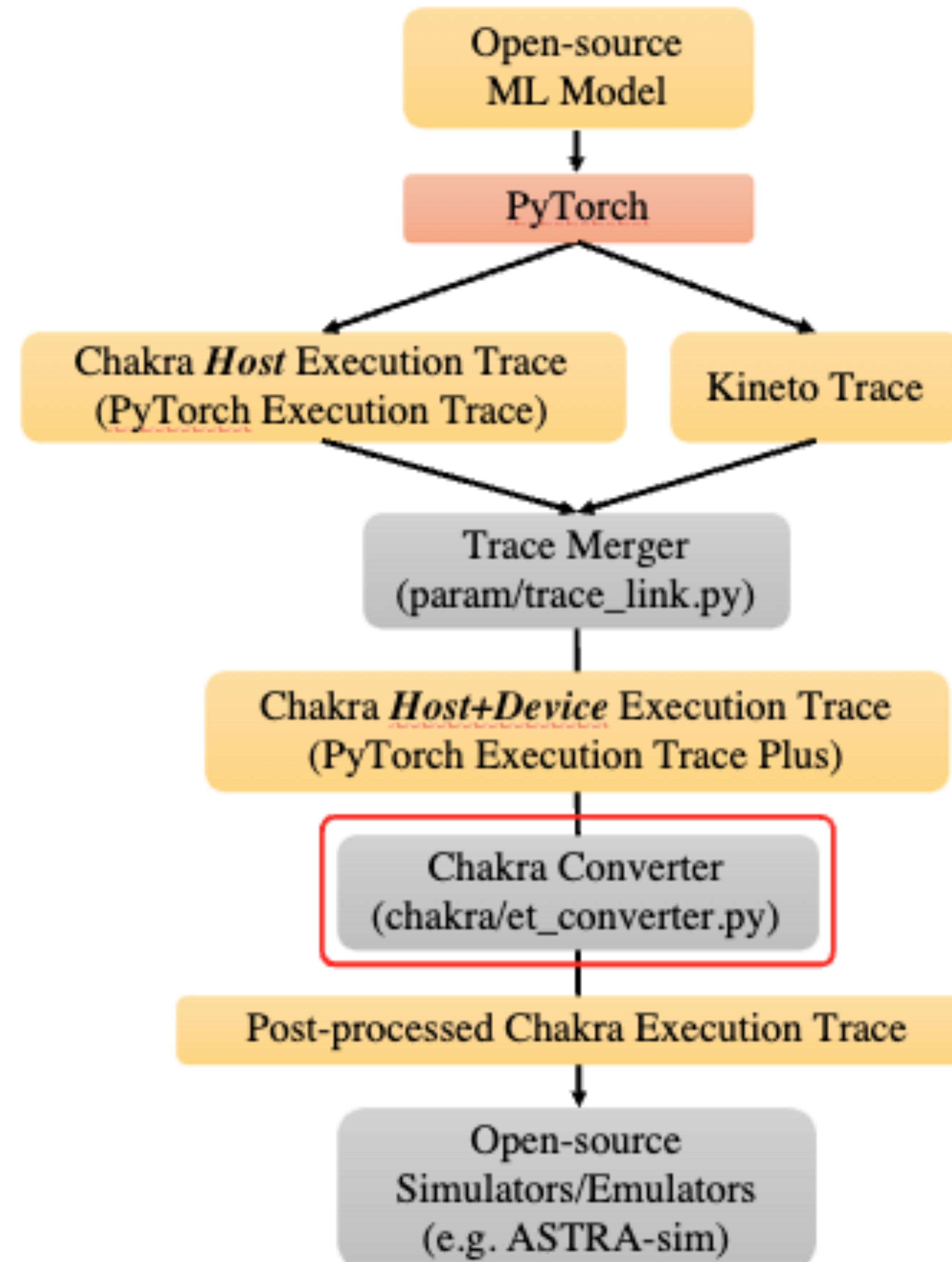
Trace Merger Internals

Merge into Chakra^{HDT} ET

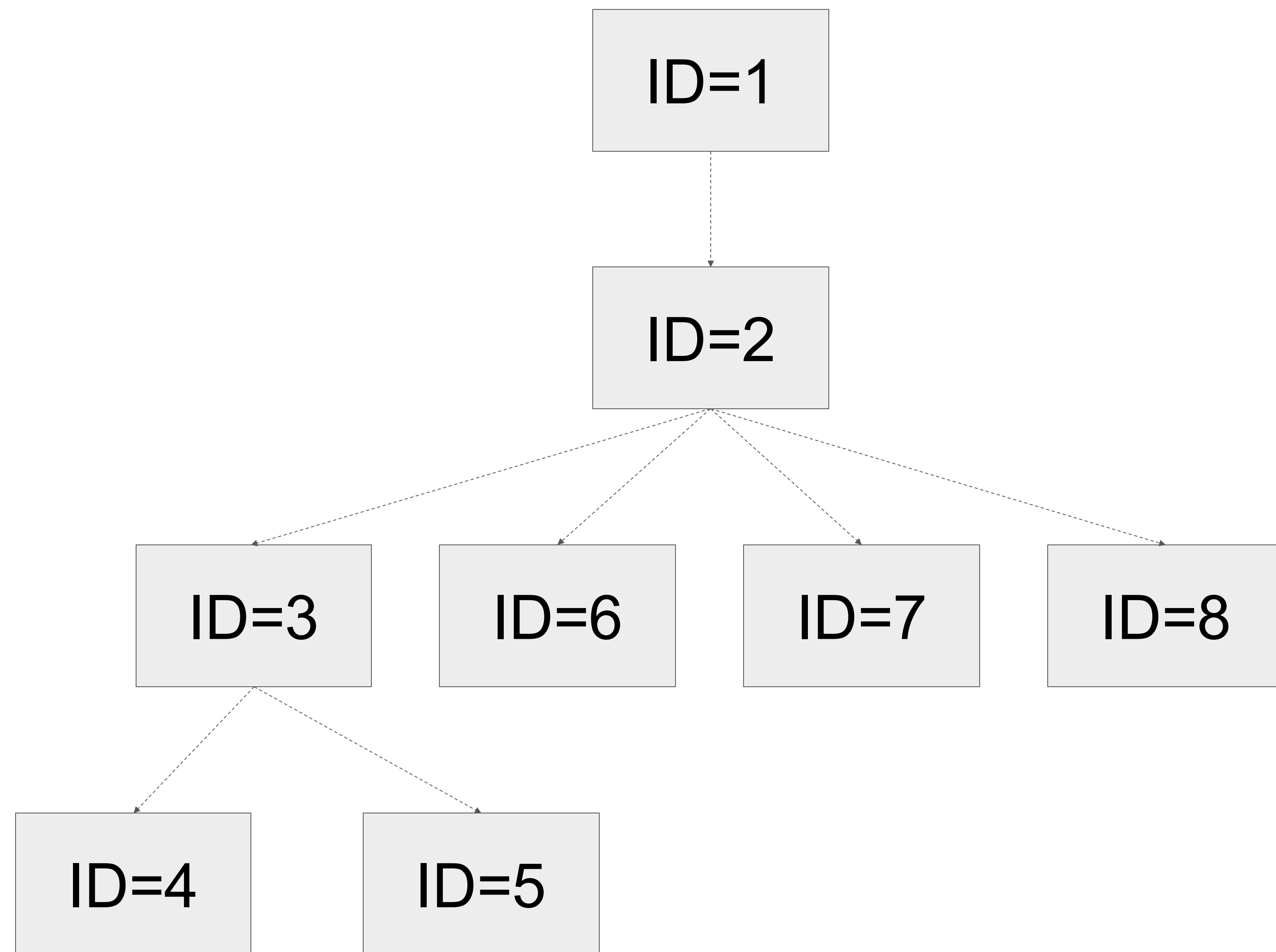


- The Chakra^{HDT} ET node has **unique operation id, timestamp, duration, dependencies, etc.**

Chakra Execution Trace Postprocessing Steps



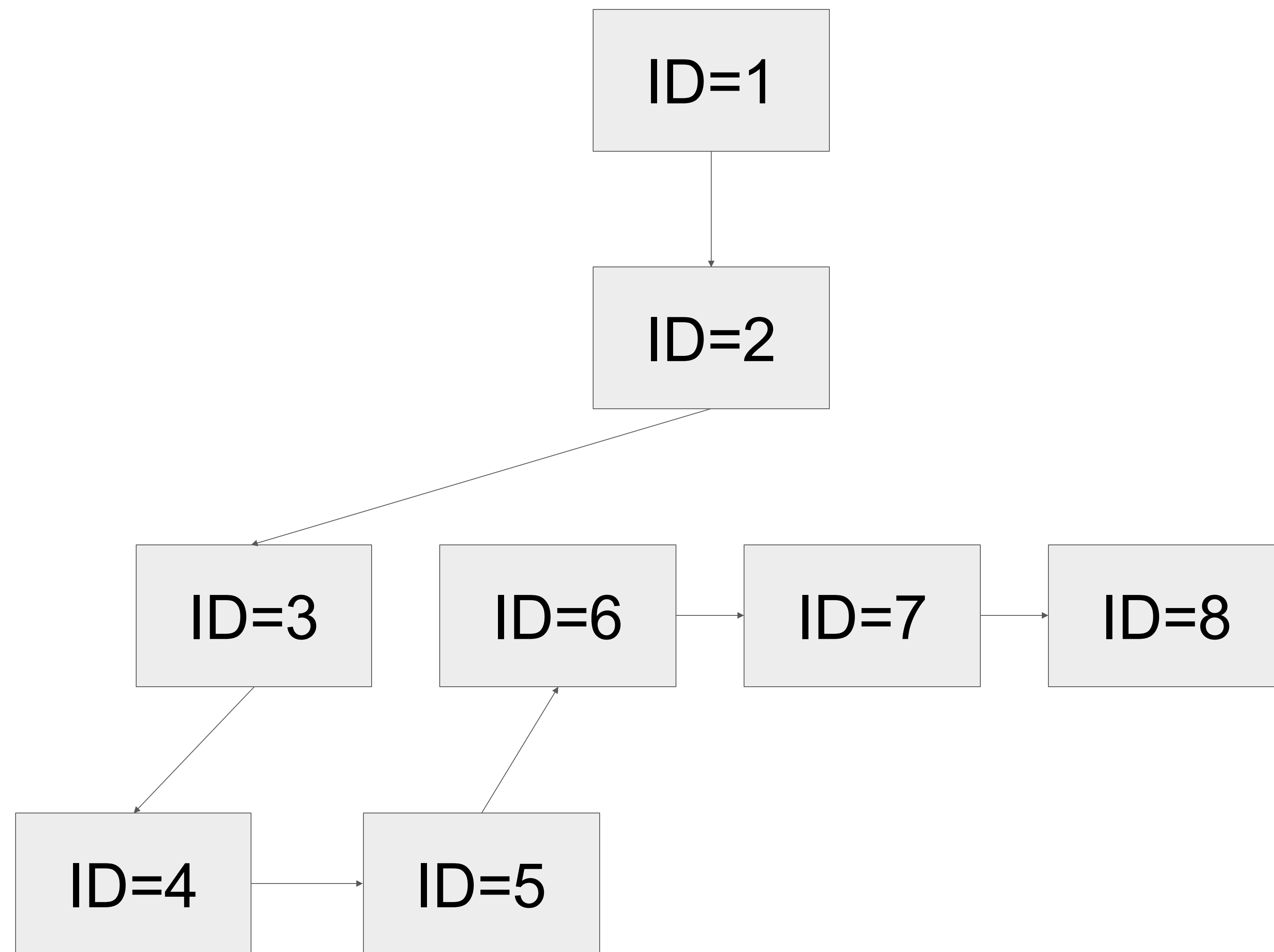
Chakra Converter Internals



-----> Control dependency

- Convert Control dependencies into Data dependencies through **Depth-First-Search**

Chakra Converter Internals

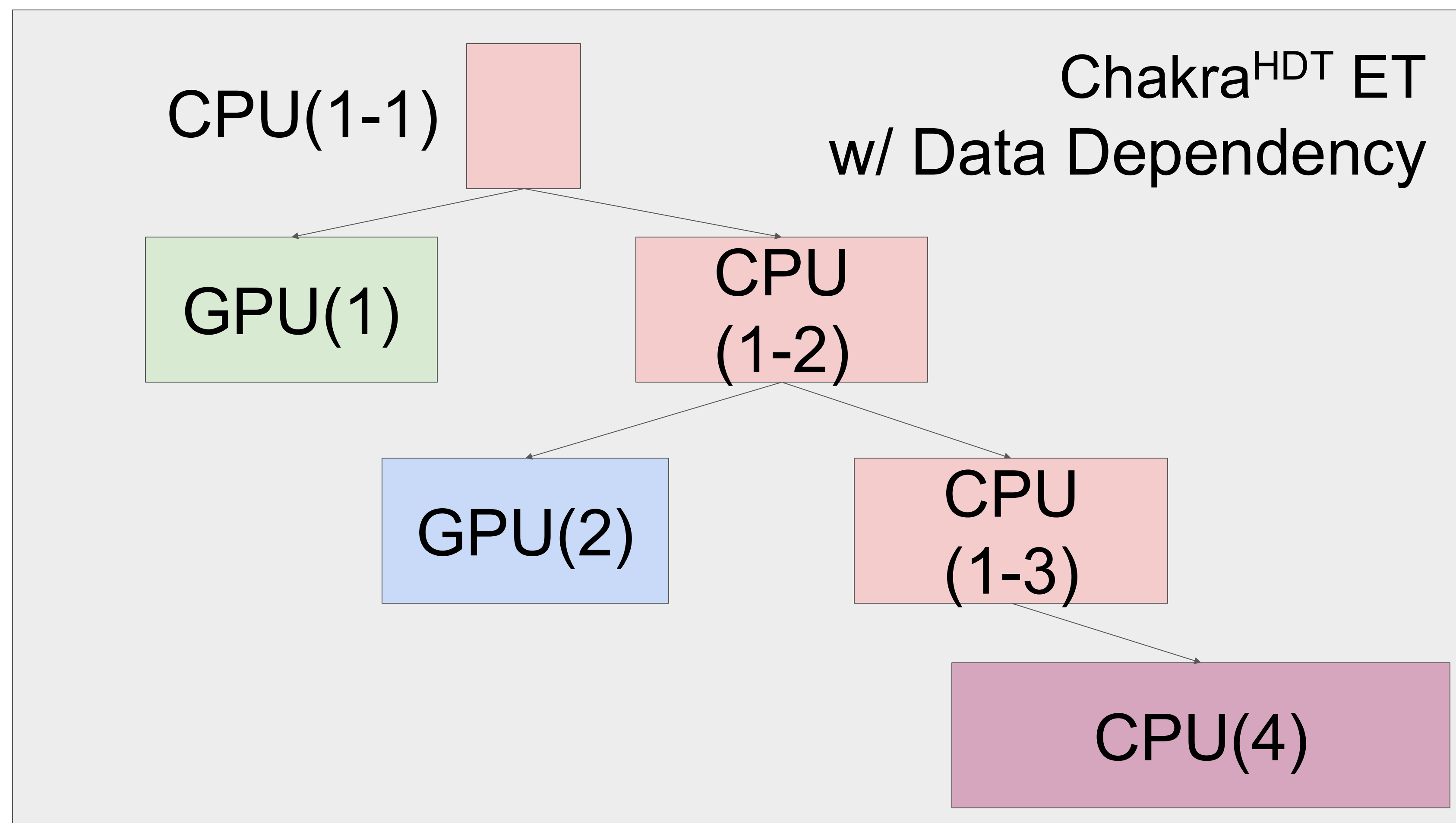
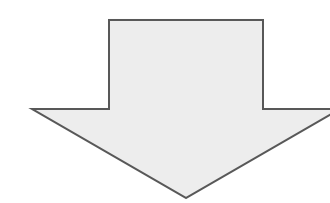
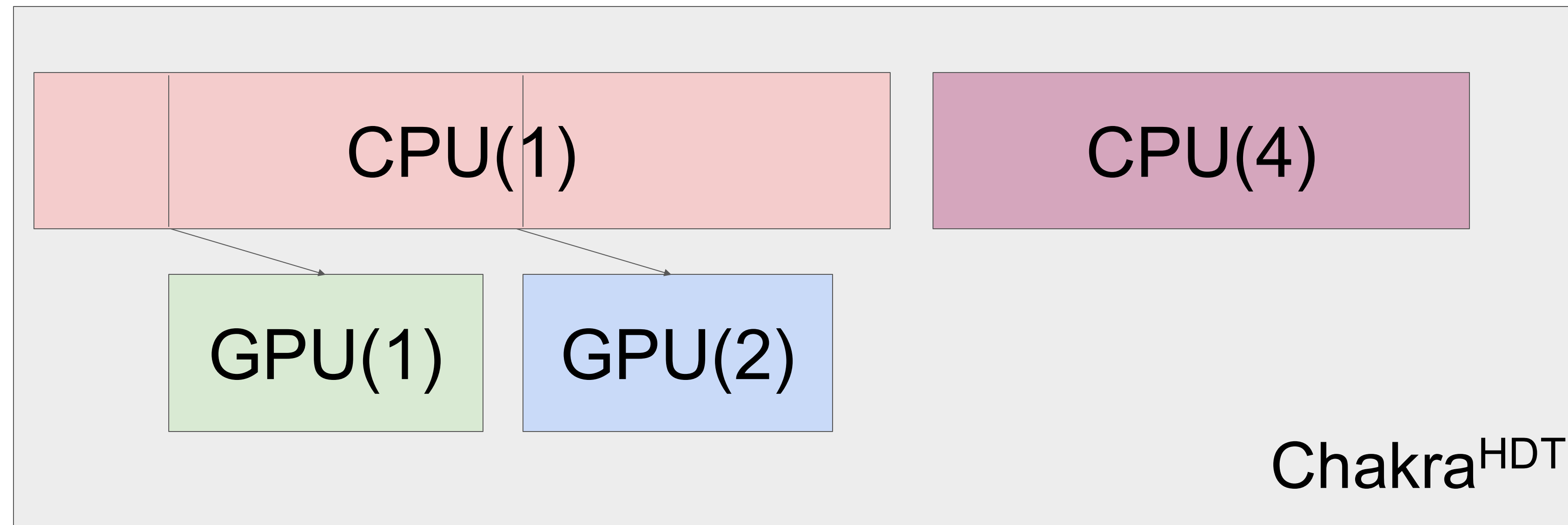


-----> Control dependency

—————> Data dependency

- Convert Control dependencies into Data dependencies through **Depth-First-Search**

Chakra Converter Internals



- Enables overlapping execution in future simulation

Chakra Ecosystem

