SOSP 2025

TrainVerify

Equivalence-Based Verification for Distributed LLM Training

Yunchi Lu[1,2], Youshan Miao[2], Cheng Tan[3,2], Peng Huang[1], Yi Zhu[2], Xian Zhang[2], Fan Yang[2]

[1]University of Michigan [2]Microsoft Reseach Asia [3]Northeastern University

LLM training is costly

LLM training requires enormous resources $\longrightarrow$ distributed training is **a must**

"Arms race" for larger models $\longrightarrow$ more resource-intensive training

Model	Training Scale	Time	Cost
Llama3	up to 16K H100 GPUs	$\sim$ 3 months	$\sim$ \$100 M
DeepSeek-V3	2K H100 GPUs	$\sim$ 2 months	$\sim$ \$6 M
PaLM	6K v4 TPUs	$\sim$ 3 months	$\sim$ \$20 M

Distributed training is notoriously error-prone

Complex training stack

- Model Code
- Parallelization
- Training Orchestration & Runtime
- Graph Lowering & Optimization
- Communication & GPU kernels
- Heterogenous HW & Drivers

LLAMA
Qwen
DeepSeek
Megatron-LM
DeepSpeed
nnScaler
PyTorch
TensorFlow
JAX
OpenXLA
tvm
Triton
NVIDIA
AMD
HUAWEI

More than millions of lines of Python / C / CUDA code

Complexity → ⚠️ distributed training is **error-prone**

Parallelization bugs

A major class of bugs that is *most elusive and costly*

Parallelization Bugs ***only*** *surface at distributed settings*

- wrong tensor/op partitioning; misused comm collectives; faulty scheduling; ...

Jeopardize the **correctness** of distributed training

Silent: often do *not* crash the training → Huge resource waste

Example: a mysterious bug in Megatron-LM

Don't blame it!

[BUG] Incorrect loss scaling in context parallel code logic

- Multi-device: Triggered when multiple GPUs are involved
- Root Cause: Incorrect loss scaling factor across ***multiple rounds*** of tensor synchronization
- Debug: Diagnosis took more than 10 days of combined efforts
- ***Hard to discover***: Bug is introduced ***over 8 months earlier***

Many models were trained without anyone realizing it.

Finding from our study

Cause	Megatron-LM	DeepSpeed	nnScaler
tensor / op partitioning	16	18	19
scheduling	4	1	4
communication	8	13	5
Total	26	28	25

All state-of-the-art distributed training frameworks encounter **parallelization bugs** in nearly every aspect of their workflow

Goal and research questions

Given the high stakes, can we **formally verify** that the parallelization logic is correct **before** distributed training starts?

While *supporting* existing systems with *moderate* verification efforts?

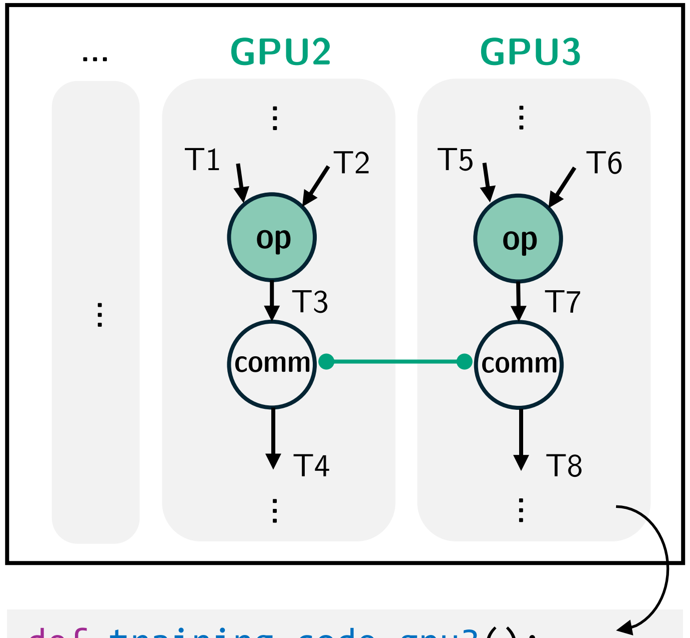
Parallel training systems: complex stack, diverse vendors, update rapidly

- *Impractical* to verify *end-to-end* training
- Prohibitive efforts to rewrite *correct-by-construction* systems

Key Insight

Principled parallelization frameworks have a well-defined ***Execution Plan***

- Typically structured as a dataflow graph (DFG)
- **Instantiated** and transformed from logical (single-device) model → **captures** parallelization *logic*
- Once an execution plan is fixed, it is **used for** driving the training




```
def training_code_gpu3():
    ...
    T7 = op(T5, T6, args)
    T8 = comm(T7, [GPU2,GPU3])
    ...
```

Parallelization correctness can be reasoned about ***at the level of Execution Plans.***

Cleaner and more **structured** → tractable verification

Contributions

Propose an Execplan-based approach to **symbolically** verify parallelization correctness

- Introduce <u>verifiability</u> into complex LLM training system
- A tool *TrainVerify* that makes this approach practical and **scalable**

Trade-off: *focused scope* to make verification tractable

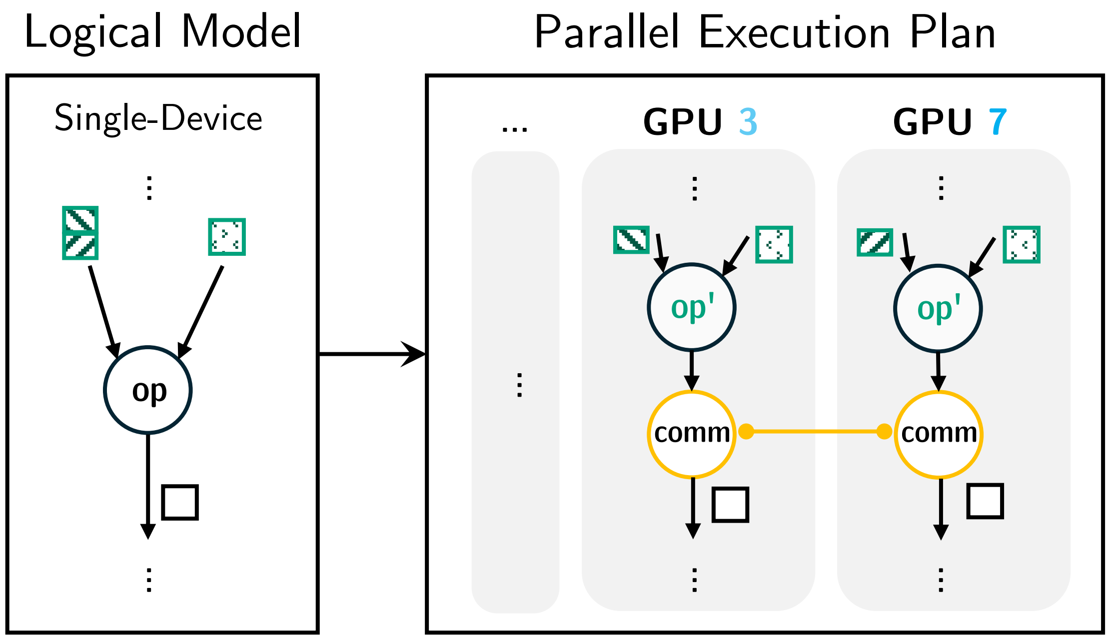
🎯 Only target correctness of parallelization *logic*

- Not other types of bugs (buffer management, GPU kernel implementation, ...)

Parallelization: Logical Model → Execplan

Parallelizing a model:

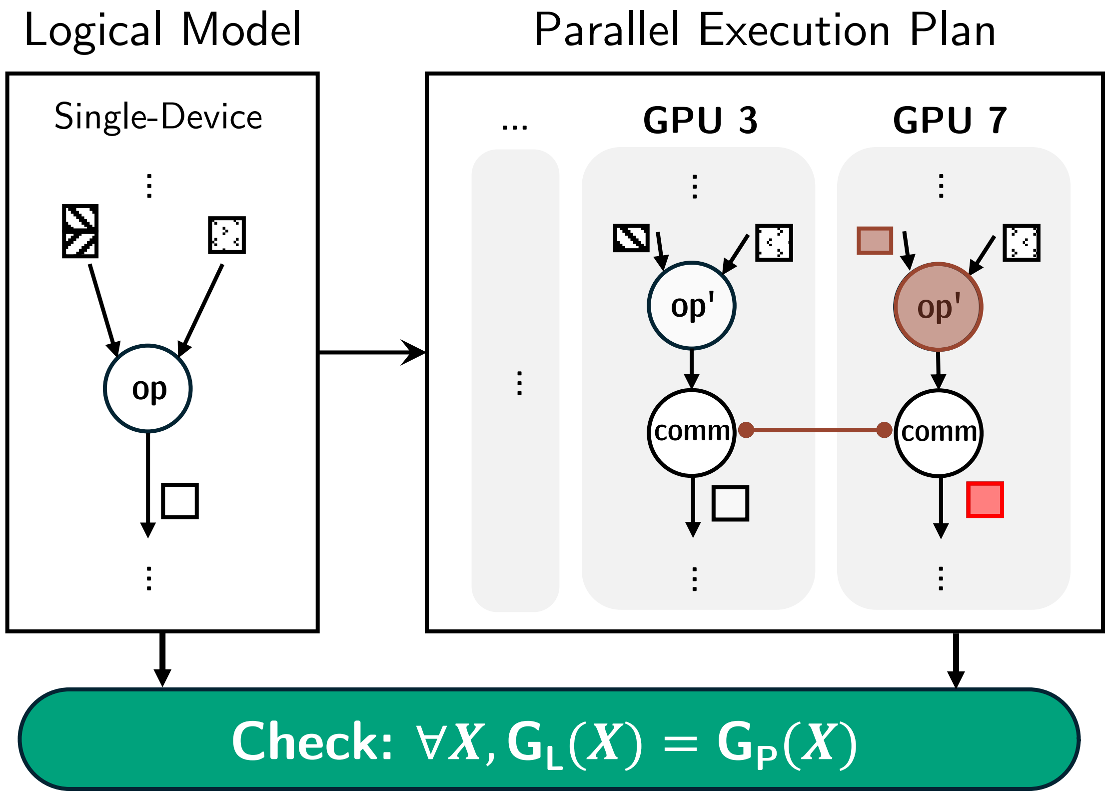
- *partition* tensors and operations
- *assign* to devices
- *synchronize* states
- *schedule* execution order



Verify **Equivalence** for Execplan

Parallelizing a model:

- *partition* tensors and operations
- *assign* to devices
- *synchronize* states
- *schedule* execution order



Buggy parallelization

- the execution plan **diverges** from the logical definition of the model

Checking ***Parallelization Equivalence*** is essential for eliminating parallelization bugs

Verify Equivalence for Execplan Symbolically

- Deep learning relies on floating point computation → **numerical drift**
- *Value-based* equivalence comparison is compromised
- Conventional differential testing subject to many false alarms

```
# Device = CPU
(1e16 + (-1e16))+ 1 = 1.0
 1e16 +((-1e16) + 1)= 0.0
```

non-associative of FP numbers

```
# Device = GPU
out1 = MatrixMultiply(A[:1,], B)
out2 = MatrixMultiply(A, B)[:1,]
Max(|out1-out2|) → 1669.2500
```

batch non-invariance

🚨 **False alarm** for correct / equivalent parallelization logic

- Factors that *amplify* numerical drift: change in computation order, parallelization, diverse GPU kernel implementation, mixed-precision training

Challenges

🎯 *Symbolically verify parallelization equivalence for execution plans*

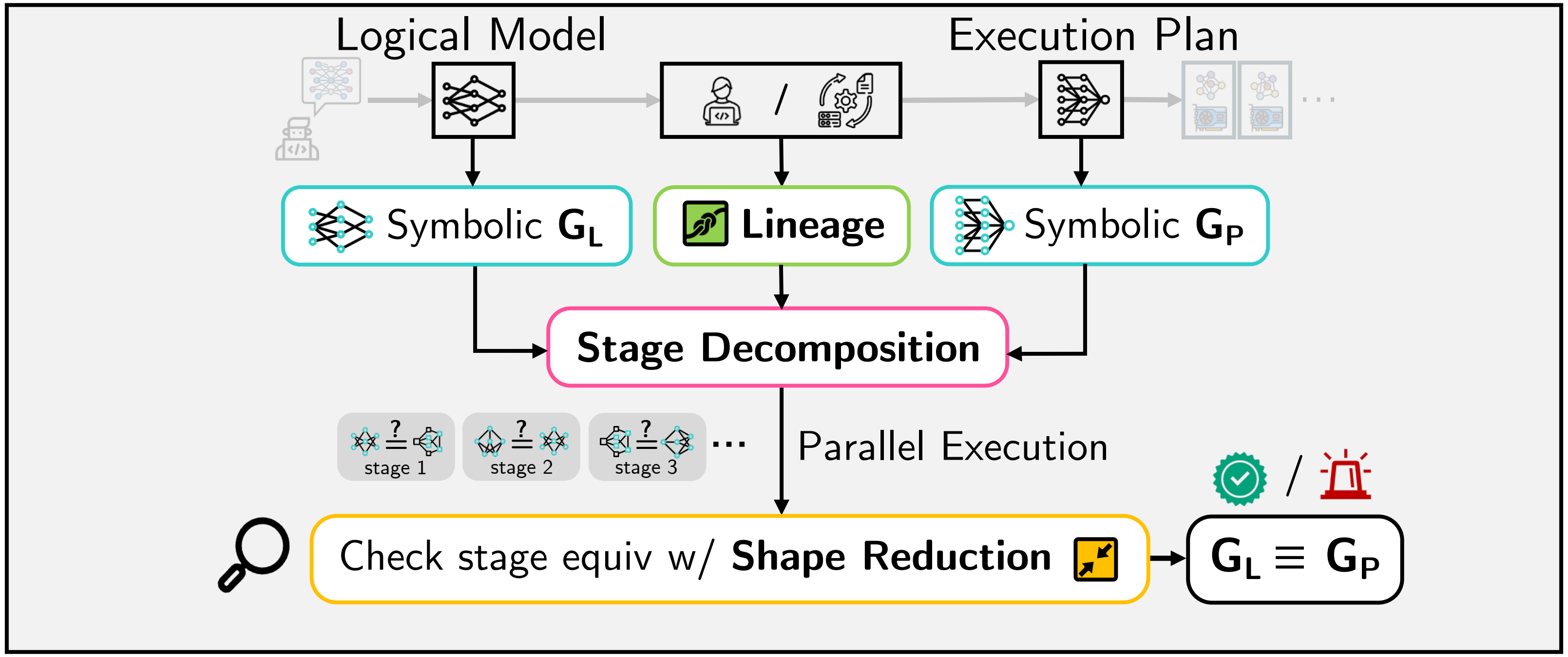
What is the **representation** to carry out the equivalence checking?

How to **scale** verification to large DNNs that have hundreds of billions of parameters?

Overview of TrainVerify

Propose symbolic dataflow graph and lineage to carry out verification

Achieve scalability through stage-parallel and shape-reduction designs



Symbolic Data Flow Graph (sDFG)

🛠️ sDFG: Constructed upon DFG of execution plans

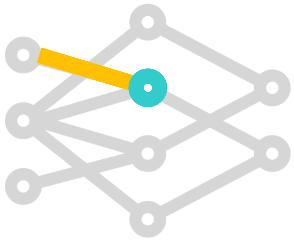
1. Preserves the same graph structure
2. PyTorch tensors → symbolic tensors
3. PyTorch operators → rewritten operators supporting symbolic tensors

🤔 ~~Tensor as *one* symbol?~~

$$\begin{bmatrix} \text{z3.Real} & \text{z3.Real} \\ \text{z3.Real} & \text{z3.Real} \end{bmatrix}$$

numpy.ndarray

Elements as symbols!



```
def symbolic_matmul(A,B):
    return A @ B
```

✅ We can construct *algebraic expressions* for model outputs.

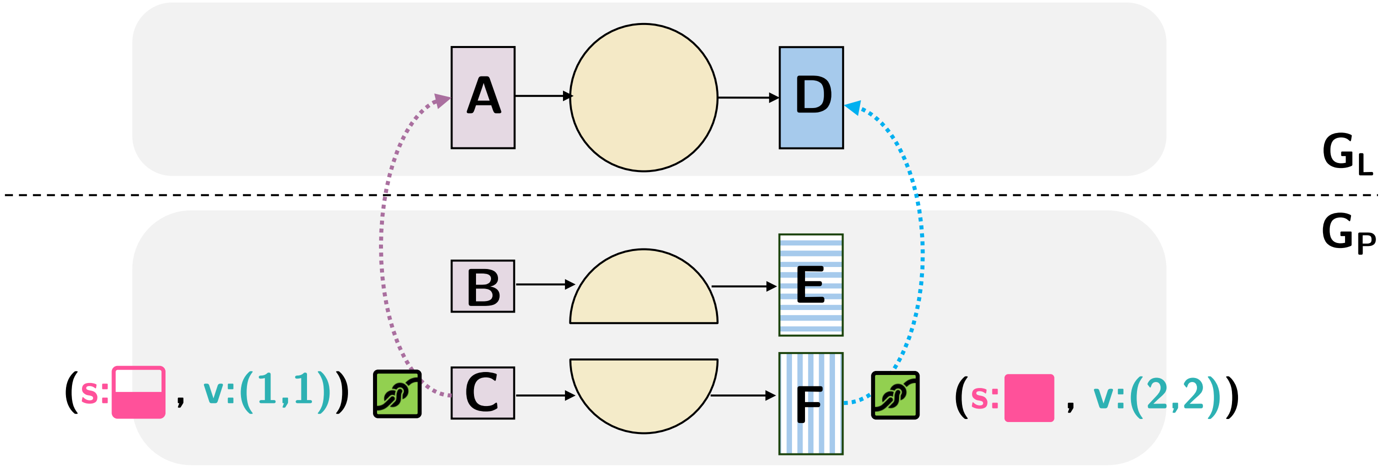
Lineage: Bridge between G_L and G_P

Crucial data structure encoding **tensor mapping** and **partition semantics**

```
From     : SubTensorID
To       : FullTensorID
SliceMap: List[Slice]
ValueMap: (VID, Count)
```

Similar Concept in multiple sys

- vTensor-pTensor in nnScaler
- DTensor in PyTorch
- …



Lineage: Bridge between G_L and G_P

Crucial data structure encoding **tensor mapping** and **partition semantics**

```
From     : SubTensorID
To       : FullTensorID
SliceMap: List[Slice]
ValueMap: (VID, Count)
```

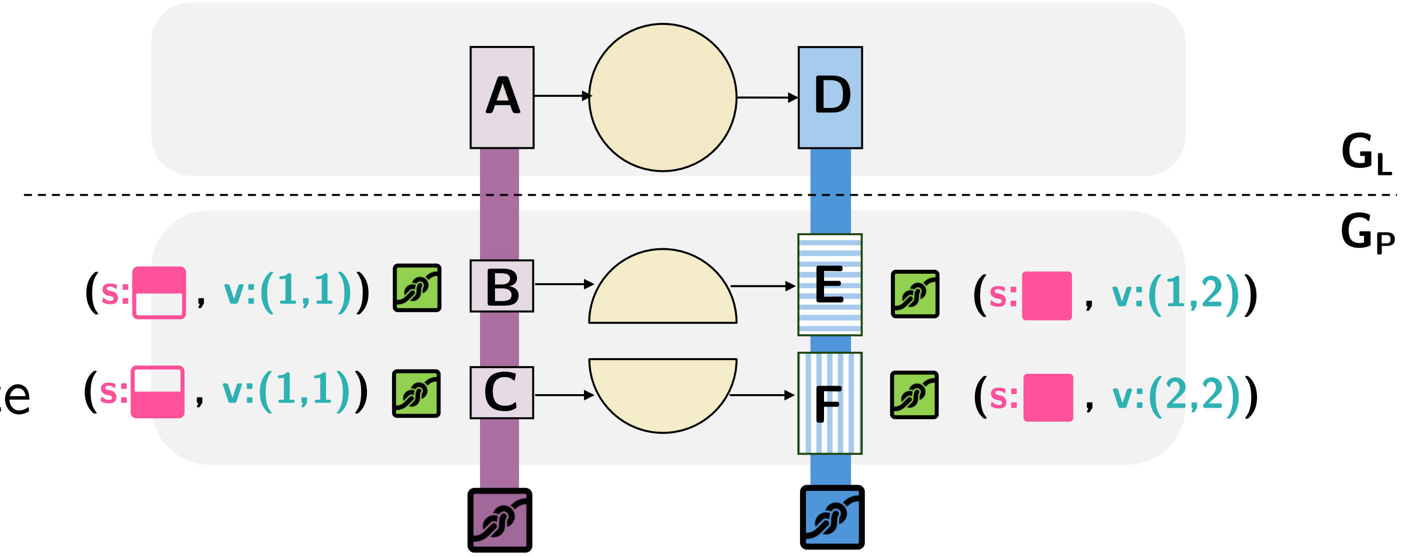
Tensor IDs follows *SSA*

Lineage ⇒ expressions

`A = concat([B,C],axis=0)`

`D = E + F`

Spec of parallelization equivalence

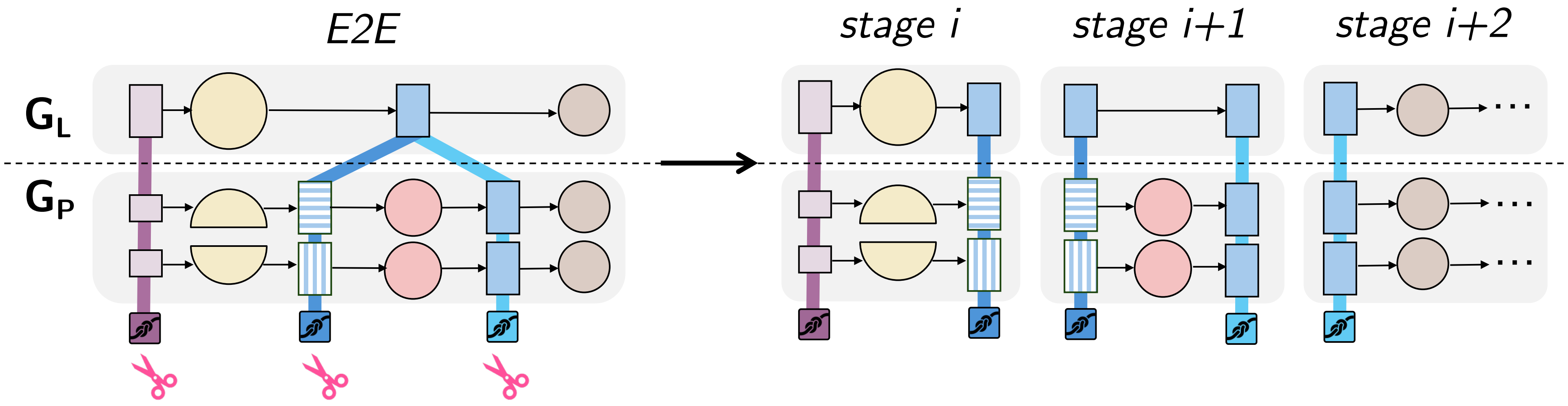


Stage Decomposition guided by lineage

🤔 Extremely **deep** architectures of LLMs

- Lengthy & nested expressions
- Intractable complexity: solver unknown / out of memory

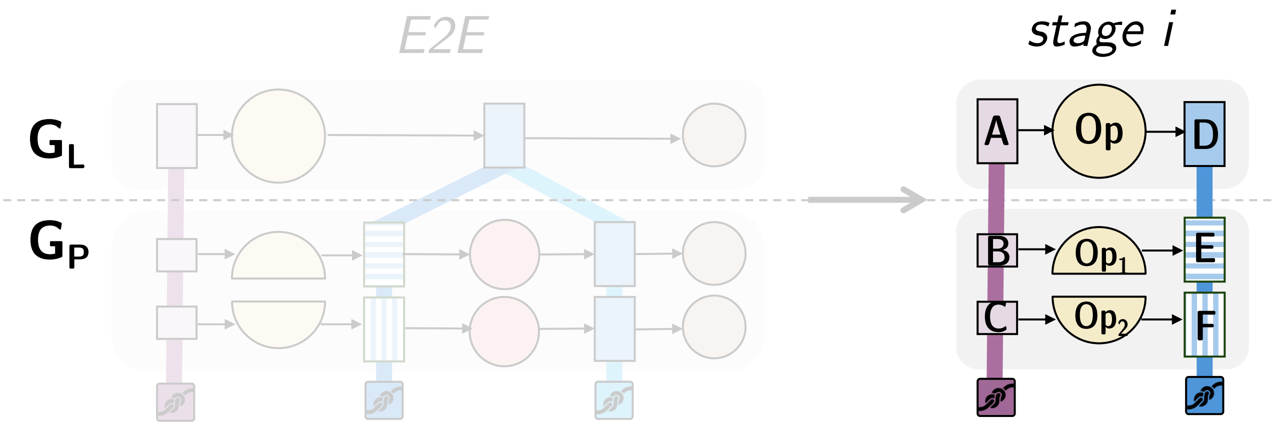
💡 **Divide & Conquer:** E2E verification → smaller, **tractable** stages



Verifying a single stage

Each stage:

- Carry lineage for input / output tensors as boundaries



Verify equivalence for a stage

Check

output equivalence

`D == E + F`

is **`AlwaysTrue`**

Given

input Equivalence

`A == concat([B,C],axis=0)`

and **operations**

$D \leftarrow Op(A)$

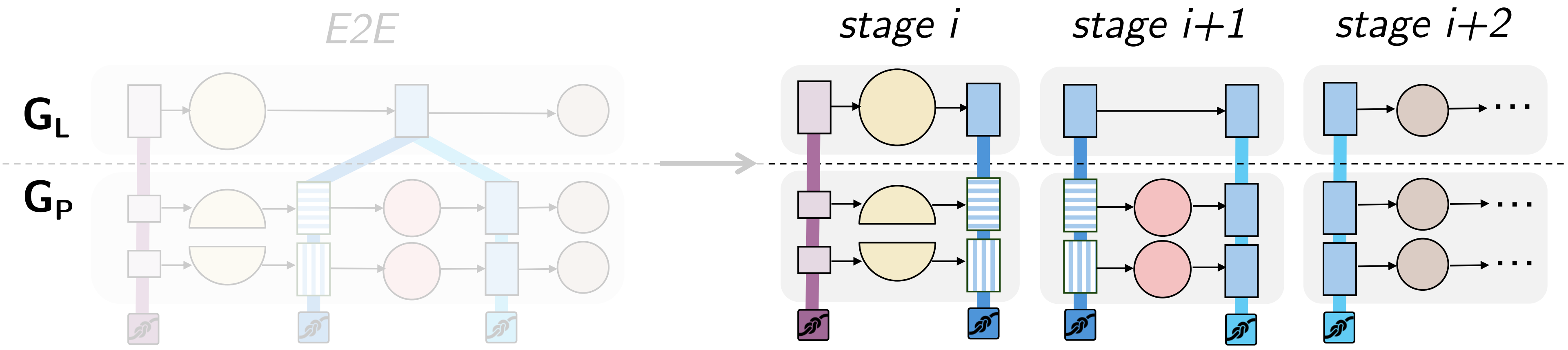
$E \leftarrow Op_1(B)$

$F \leftarrow Op_2(C)$

Verifying all stages

Independently verfiable; executed in **parallel**

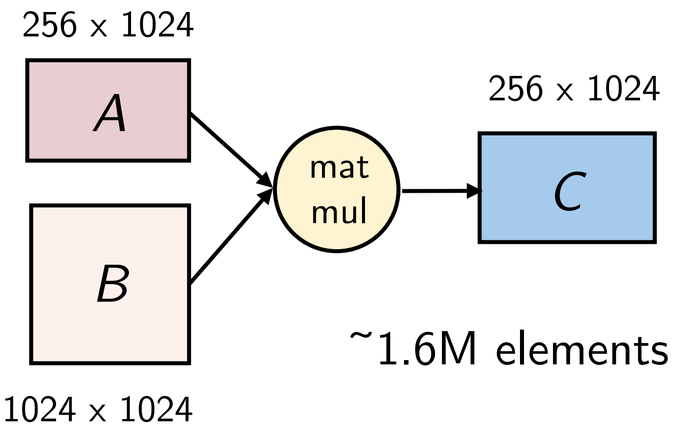
- All stages pass $\Longrightarrow$ **End to end equivalence** between G_L and G_P
- A failed stage $\Longrightarrow$ Localized equivalence violation



Shape Reduction

🤔 Original shape → Too many elements

- Tensors with symbolic real as elements



Leverage **repetition** in DNN operators

- Same operation applied repeatedly across multiple sub-tensors

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \times \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,1} & b_{3,2} & b_{3,3} \end{bmatrix} = \begin{bmatrix} c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,1} & c_{2,2} & c_{2,3} \\ c_{3,1} & c_{3,2} & c_{3,3} \end{bmatrix}$$

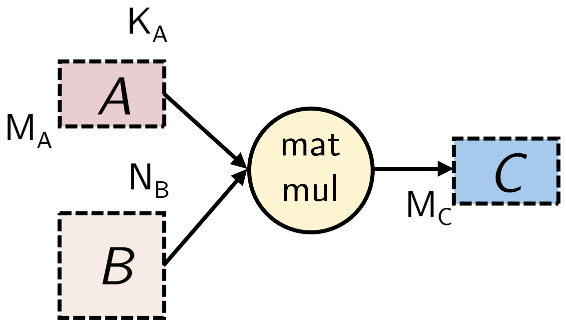
$$c_{1,1} = a_{1,1} \cdot b_{1,1} + a_{1,2} \cdot b_{2,1} + a_{1,3} \cdot b_{3,1}$$

$$c_{2,2} = a_{2,1} \cdot b_{1,2} + a_{2,2} \cdot b_{2,2} + a_{2,3} \cdot b_{3,2}$$

Shape Reduction

💡 Reduce redundancy with shape reduction

- Verify equivalence on the same sDFGs with **tensor shapes reduced**

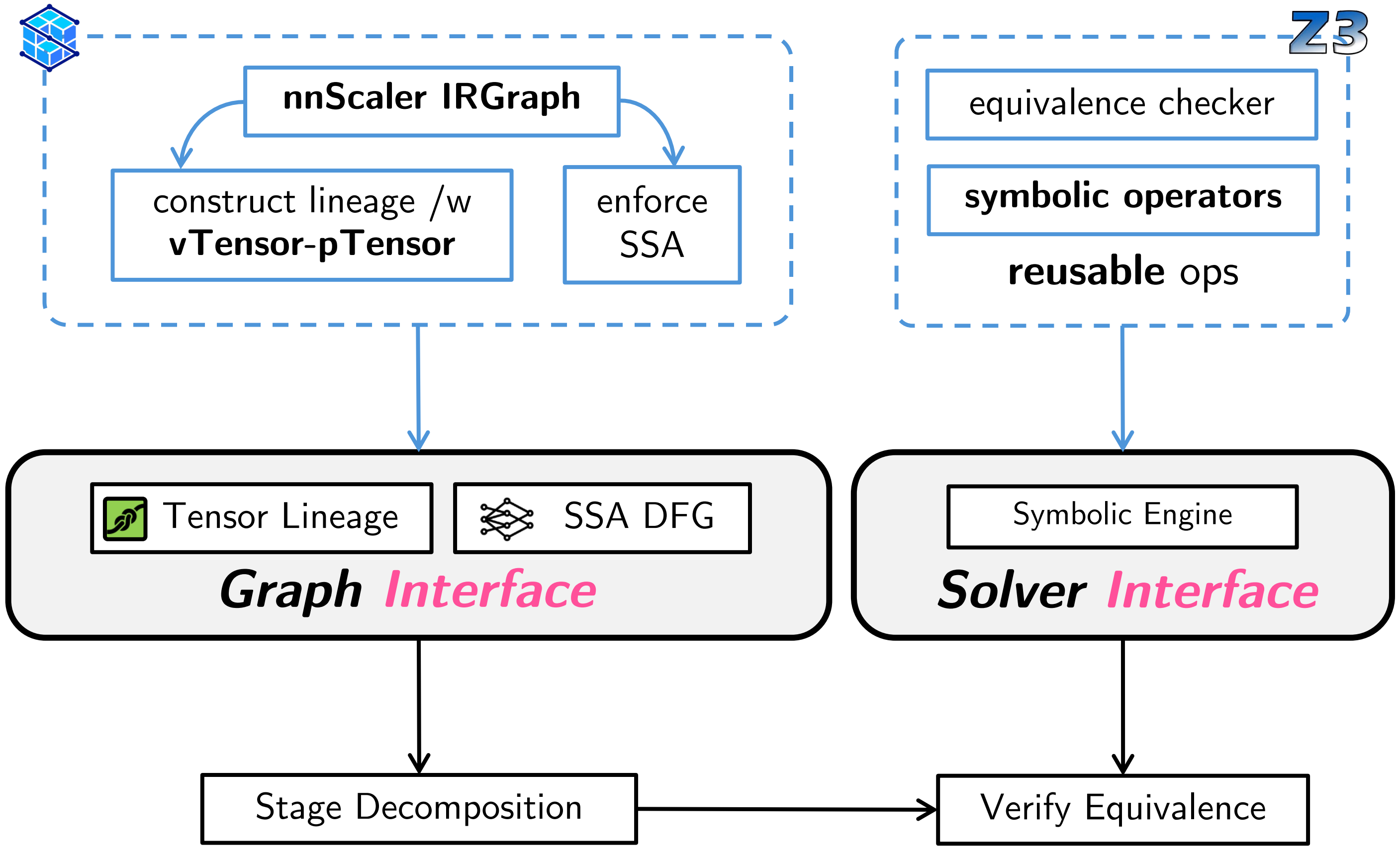


Preserve structural and functional **properties**

- **Following** shape alignment & operator intactness **constraints**

Verified equivalence of shape-reduced models faithfully extends to the original

Implementation w/ Modular Design



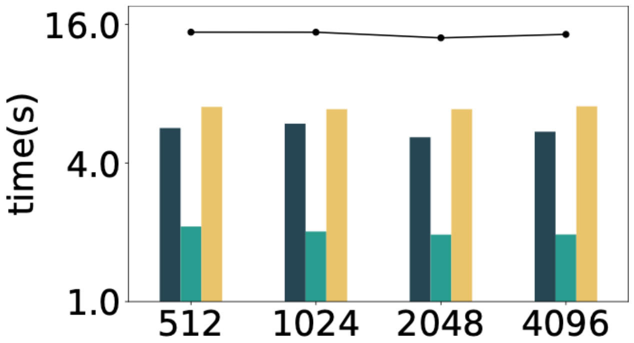
Verify Large-Scale LLM Parallelization

Verification Setup: **single** machine with a 32-core CPU and 1.3 TB memory

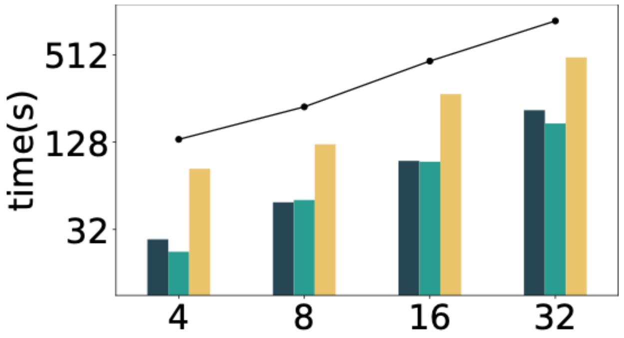
ID	Model	Size	GPUs	Time
L1	Llama3	8B	512	0.2h
L2		70B	512	2.4h
L3		**405B**	**8192**	**8.0h**
D1	DeepSeek-V3	16B	128	0.4h
D2		236B	512	2.4h
D3		**671B**	**2048**	**9.0h**

- TrainVerify **scales to the largest trainings**
- Can run *asynchronously* alongside training

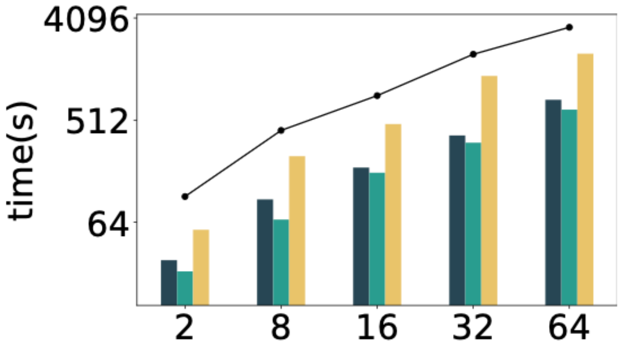
Scalability Trends



Global Batch Size
Hidden Dimension
Sequence Length



Num Layers



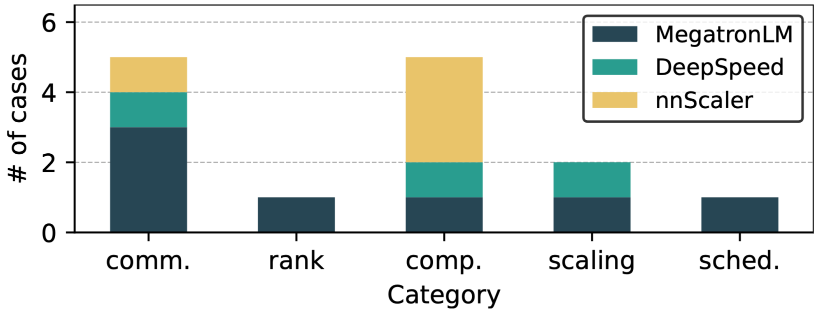
Data Parallelism Degree
Tensor Parallelism Degree

Verification cost is invariant to original tensor shapes.

Verification cost is linear to layers and parallelization degrees.

Eliminate Broad Categories of Bugs

Successfully detect *reproduced* parallelization bugs



Expose *new* bugs

C1: Sharding a non-partitionable dimension

C2: Dangling tensors in nnScaler IRGraph

C3: Violation of tensor SSA in transformation

TrainVerify

Symbolic DFG

Tensor Lineage

IR

Correctness

Distributed LLM Training

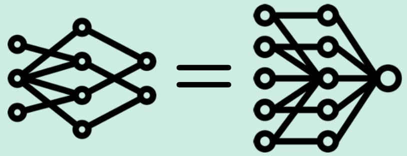
Scope: Model Parallelization

Efficient Checking

Stage Decomposition

Shape Reduction

Parallelization Equivalence



Execution Plans

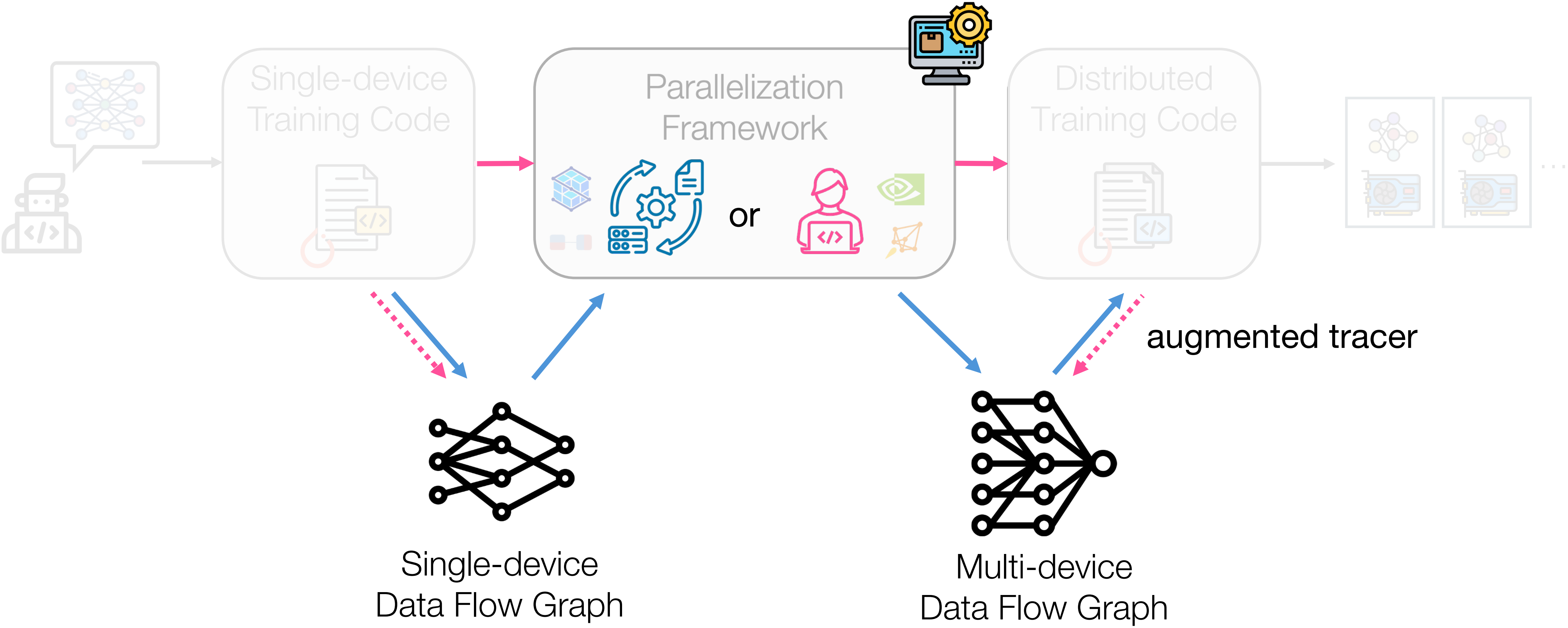
https://github.com/verify-llm/TrainVerify

Scalable

Eliminate Bugs

Backup Slides

Data Flow Graph Availability



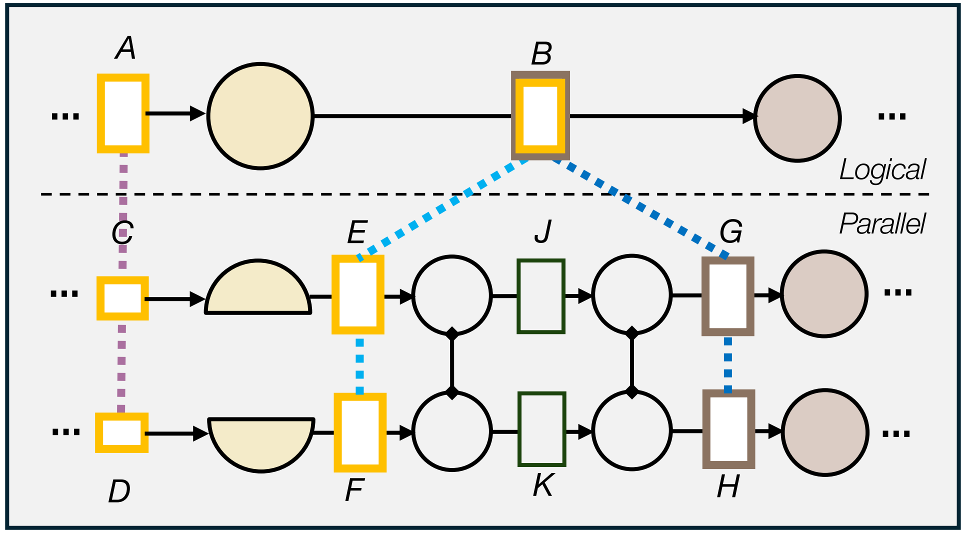
A Training Iteration

```python
for epoch in range(num_epochs):
    model.train()

    for x, target in train_loader:
        optimizer.zero_grad()      # reset gradients
        y = model(x)               # forward
        loss = criterion(y, target) # loss
        loss.backward()            # backward
        optimizer.step()           # update parameters

    print(...) # validation, logging, checkpointing
```

TrainVerify focuses on one iteration to leverage **loop homogeneity**.

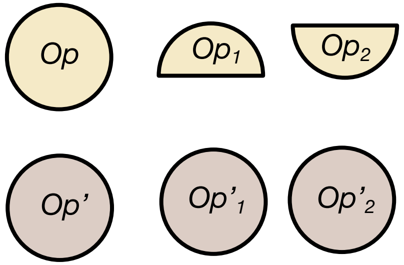
For inhomogeneous loops, we can run verification for each of them.

Operator Alignment



Operators appearing in both logical and parallelized graphs bear **source code location** as **beacon for alignment**.

Operator Alignment → Tensor Lineage Alignment



Source code location: # A/B.model, line 24, $X_2 = Op(X_1)$

Source code location: # A/B.model, line 25, $X_3 = Op(X_2)$

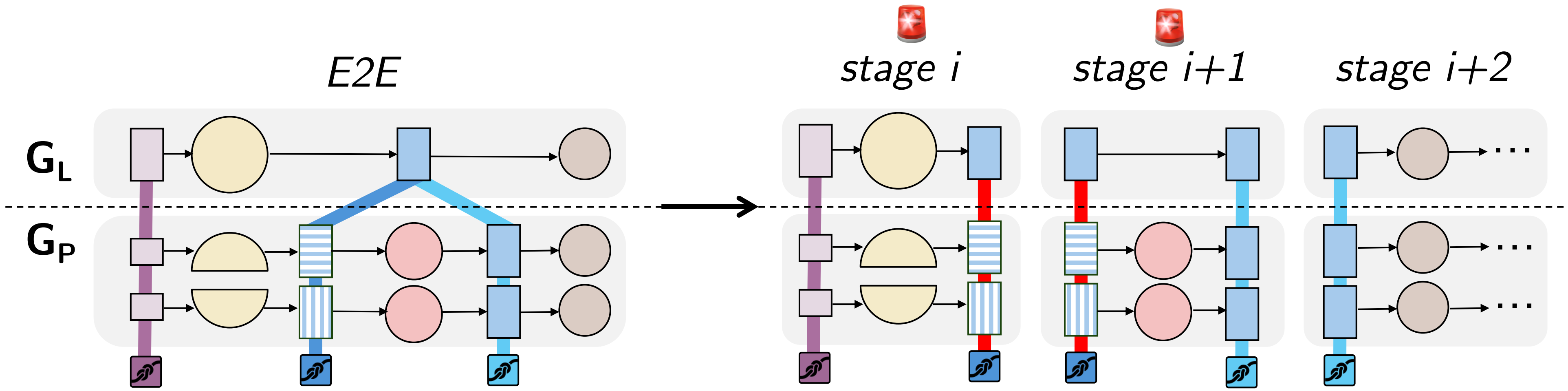
Lineage as Spec for PE

```
From     : SubTensorID
To       : FullTensorID
SliceMap: List[Slice]
ValueMap: (VID, Count)
```

Lineage

What if lineage is wrong?

- Possible: false positive (false alarm)
- Guarantee: no false negative (false pass)



Expressiveness of symbolics

Symbolics can ***not:***

- Serve as array indexing: arr[$\mathbf{x}$]
- Express index as output: $\mathbf{x}$ = argmax($\cdot$)

Affect a small number of ops

Solution:

- Need tricks in rewriting symbolic operator
- Use *uninterpretd function* to encode tensor algebraic properties, make it adequate for passing equivalence checks for **practical** parallelization