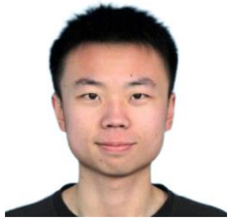
# TorchOpt: An Efficient Library for Differentiable Optimization


Jie Ren[*], Xidong Feng[*], <u>Bo Liu</u>[*], Xuehai Pan[*],




Yao Fu, Luo Mai[†], Yaodong Yang[†]



MetaOPT Team

# TorchOpt

- Architecture Overview

**Unified and expressive differentiable optimization programming**

Optimizer | MetaOptimizer | FuncOptimizer | Implicit Differentiation | Zero-order Differentiation

Gradient Transformation
- Optimizer: SGD, Adam, AdamW, RMSProp
- Hook: zero_nan_hook
- Clip: clip_grad_norm
- Combine: chain, chain_flat

Linear Solver
- Conjugate Gradient
- Neumann Series

Debugging Tool
- Gradient Graph Visualization

TransformUpdateFn | TransformInitFn | Schedule | Linalg

**High-performance and distributed execution runtime**

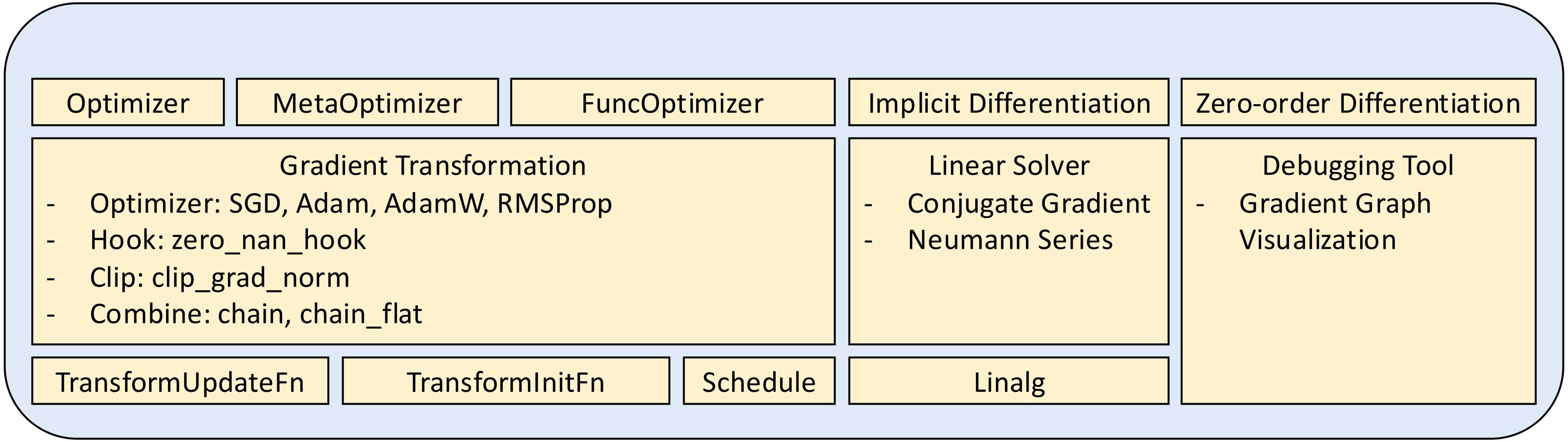
Accelerated Operator
- Symbolic Elimination
- C++ OpenMP / CUDA

Optimized PyTree Utilities
- Cache-friendly C++ Binding
- Python Built-in Type Support

Distributed
- Distributed Autograd
- Auto Parallelization
- Heterogenous Computational Graph

# TorchOpt

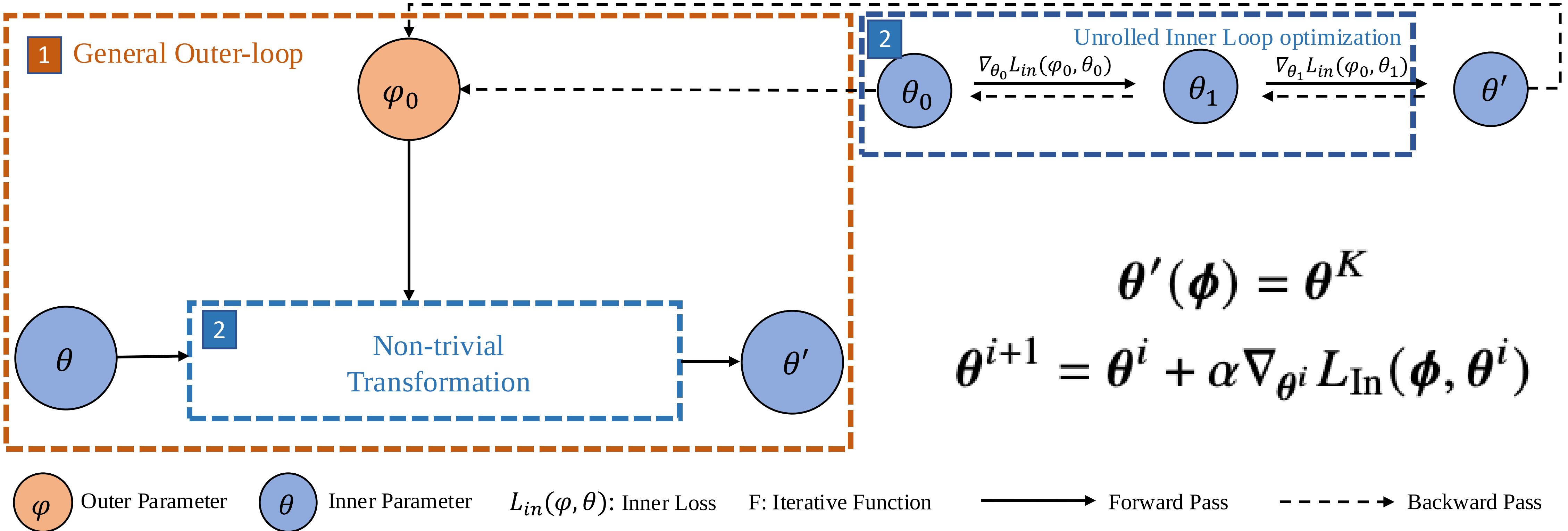
- Unified and expressive differentiable optimization programming



# Unified and expressive differentiation mode for differentiable optimization

- Explicit Gradient Differentiation

$$\text{inner loop: } \boldsymbol{\theta} \rightarrow \boldsymbol{\theta}'(\boldsymbol{\phi}), \qquad \text{outer loop: } \boldsymbol{\phi}^{\star} = \arg\min_{\boldsymbol{\phi}} L\left(\boldsymbol{\theta}'(\boldsymbol{\phi})\right)$$




$$\boldsymbol{\theta}'(\boldsymbol{\phi}) = \boldsymbol{\theta}^{K}$$

$$\boldsymbol{\theta}^{i+1} = \boldsymbol{\theta}^{i} + \alpha \nabla_{\boldsymbol{\theta}^{i}} L_{\text{In}}(\boldsymbol{\phi}, \boldsymbol{\theta}^{i})$$

# Unified and expressive differentiation mode for differentiable optimization

- Explicit Gradient Differentiation

```python
# Functional API
opt = torchopt.adam()
# Define meta and inner parameters
meta_params = ...
fmodel, params = make_functional(model)
# Initialize optimizer state
state = opt.init(params)

for iter in range(iter_times):
    loss = inner_loss(fmodel, params, meta_params)
    grads = torch.autograd.grad(loss, params)
    # Apply non-inplace parameter update
    updates, state = opt.update(grads, state, inplace=False)
    params = torchopt.apply_updates(params, updates)

loss = outer_loss(fmodel, params, meta_params)
meta_grads = torch.autograd.grad(loss, meta_params)
```

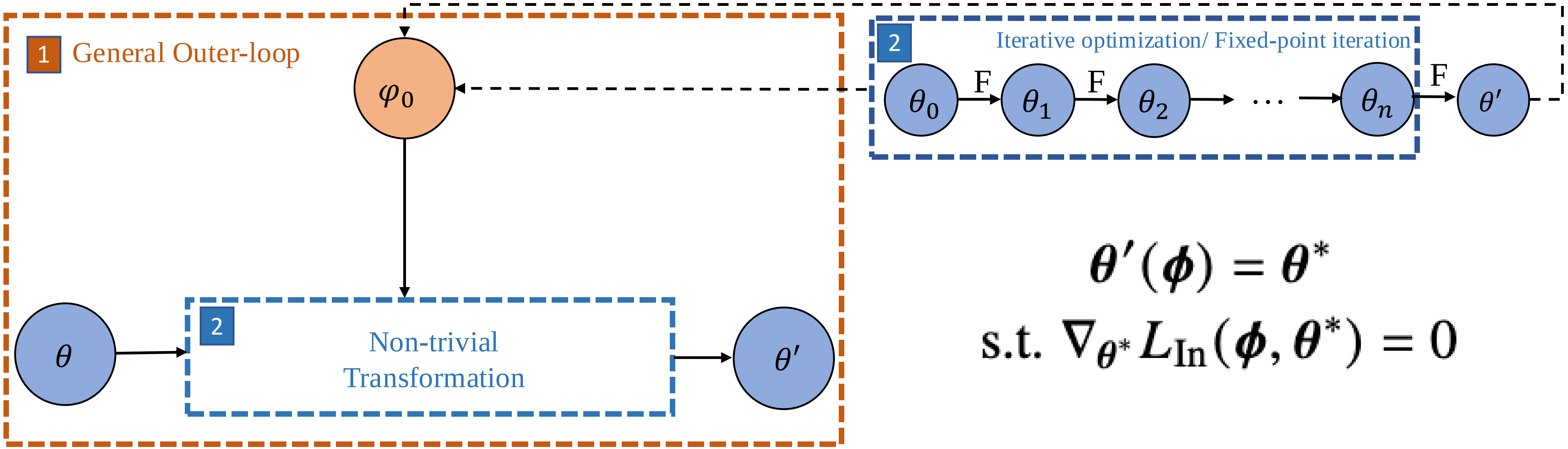
```python
# OOP API
# Define meta and inner parameters
meta_params = ...
model = ...
# Define differentiable optimizer
opt = torchopt.MetaAdam(model)

for iter in range(iter_times):
    # Perform the inner update
    loss = inner_loss(model, meta_params)
    opt.step(loss)

loss = outer_loss(model, meta_params)
loss.backward()
```

Listing 1: `TorchOpt` code snippet for explicit gradient.

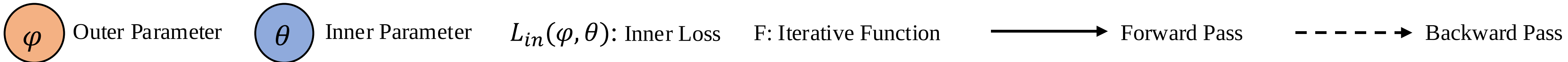
# Unified and expressive differentiation mode for differentiable optimization

- Implicit Gradient Differentiation

$$\text{inner loop: } \boldsymbol{\theta} \longrightarrow \boldsymbol{\theta}'(\boldsymbol{\phi}), \qquad \text{outer loop: } \boldsymbol{\phi}^{\star} = \arg\min_{\boldsymbol{\phi}} L\left(\boldsymbol{\theta}'(\boldsymbol{\phi})\right)$$




$$\boldsymbol{\theta}'(\boldsymbol{\phi}) = \boldsymbol{\theta}^{*}$$
$$\text{s.t. } \nabla_{\boldsymbol{\theta}^{*}} L_{\text{In}}(\boldsymbol{\phi}, \boldsymbol{\theta}^{*}) = 0$$



# Unified and expressive differentiation mode for differentiable optimization

- Implicit Gradient Differentiation

```python
# Functional API for implicit gradient
def stationary(params, meta_params, batch, labels):
    # Stationary condition construction
    ...
    return stationary condition

@torchopt.diff.implicit.custom_root(stationary)
def solve(params, meta_params, batch, labels):
    # Forward optimization process
    ...
    return optimal_params
```

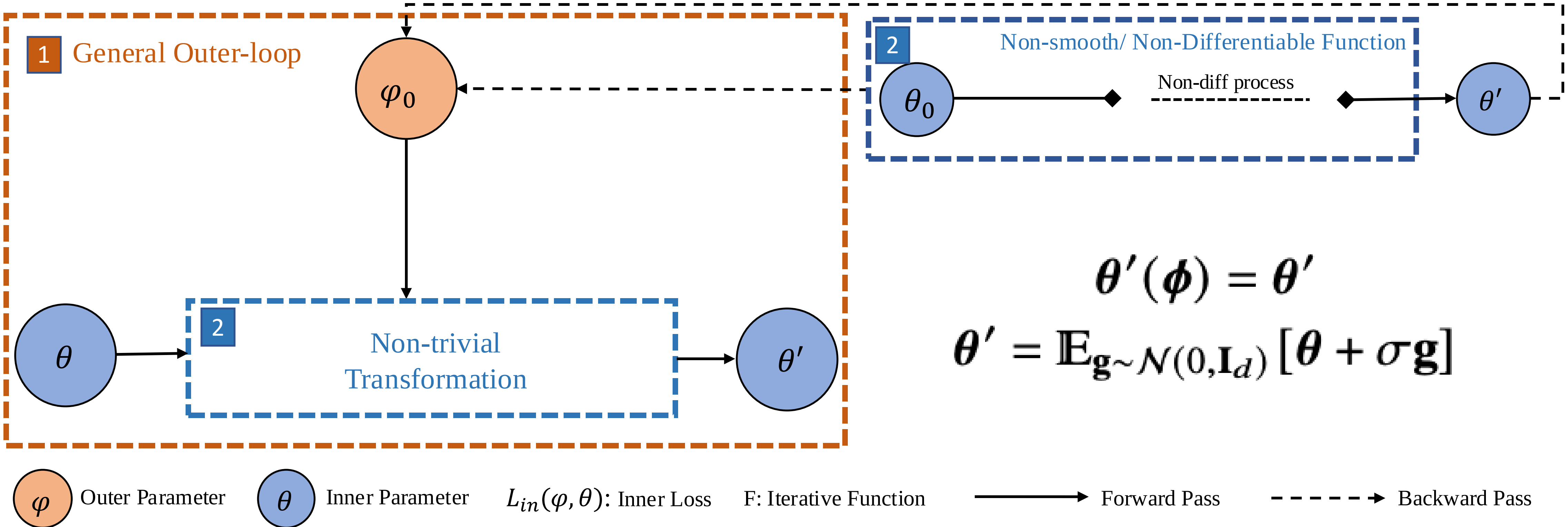
```python
# OOP API
class Module(torchopt.nn.ImplicitMetaGradientModule):
    def __init__(self, meta_module, ...):
        ...
    def forward(self, x):
        # Forward process
        ...
    def optimality(self, batch, labels):
        # Stationary condition construction
        ...
    def solve(self, batch, labels):
        # Forward optimization process
        ...
        return self
```

Listing 2: `TorchOpt` code snippet for implicit gradient.

# Unified and expressive differentiation mode for differentiable optimization

- Zero-order Gradient Differentiation

$$\text{inner loop: } \boldsymbol{\theta} \rightarrow \boldsymbol{\theta}'(\boldsymbol{\phi}), \qquad \text{outer loop: } \boldsymbol{\phi}^\star = \arg\min_{\boldsymbol{\phi}} L\left(\boldsymbol{\theta}'(\boldsymbol{\phi})\right)$$




$$\boldsymbol{\theta}'(\boldsymbol{\phi}) = \boldsymbol{\theta}'$$

$$\boldsymbol{\theta}' = \mathbb{E}_{\mathbf{g}\sim\mathcal{N}(0,\mathbf{I}_d)}\left[\boldsymbol{\theta} + \sigma\mathbf{g}\right]$$

# Unified and expressive differentiation mode for differentiable optimization

- Zero-order Gradient Differentiation

```python
# Functional API
# Customize the noise sampling function in ES
def sample(params, batch, labels, shape):
    ...
    return sample_noise
# Specify the method and parameter of ES
@torchopt.diff.zero_order(method, sample)
def forward(params, batch, labels):
    # Forward process
    return output
```

```python
# OOP API
class ESModule(torchopt.nn.ZeroOrderGradientModule):
    def sample(self, batch, labels, sample_shape):
        # Customize the noise sampling function in ES
        ...
        return sample_noise
    def forward(self, batch, labels):
        # Forward process
        ...
        return output
```

Listing 3: TorchOpt code snippet for zero-order differentiation.

# TorchOpt

- High-performance and distributed execution runtime

**Accelerated Operator**
- Symbolic Elimination
- C++ OpenMP / CUDA

**Optimized PyTree Utilities**
- Cache-friendly C++ Binding
- Python Built-in Type Support

**Distributed**
- Distributed Autograd
- Auto Parallelization
- Heterogenous Computational Graph

# High-performance and distributed execution runtime

- OpTree: Optimized PyTree Utilities
  - Memory Efficient and High-Performance (**20x faster** than `torch.utils._pytree`)
  - Cache Friendly (`absl::InlinedVector`)
  - Built-in support for common Python containers (**2x faster** than `jax.tree_util`)
    - `tuple`, `list`, `dict`, `namedtuple`
    - `OrderedDict`, `defaultdict`, `deque`
  - Support both "None is leaf" (default of PyTorch) and "None is node" (default of JAX)
    - Friendly for tensor container extraction:

      `nn.Module._parameters: Dict[str, Optional[Tensor]]`

# High-performance and distributed execution runtime

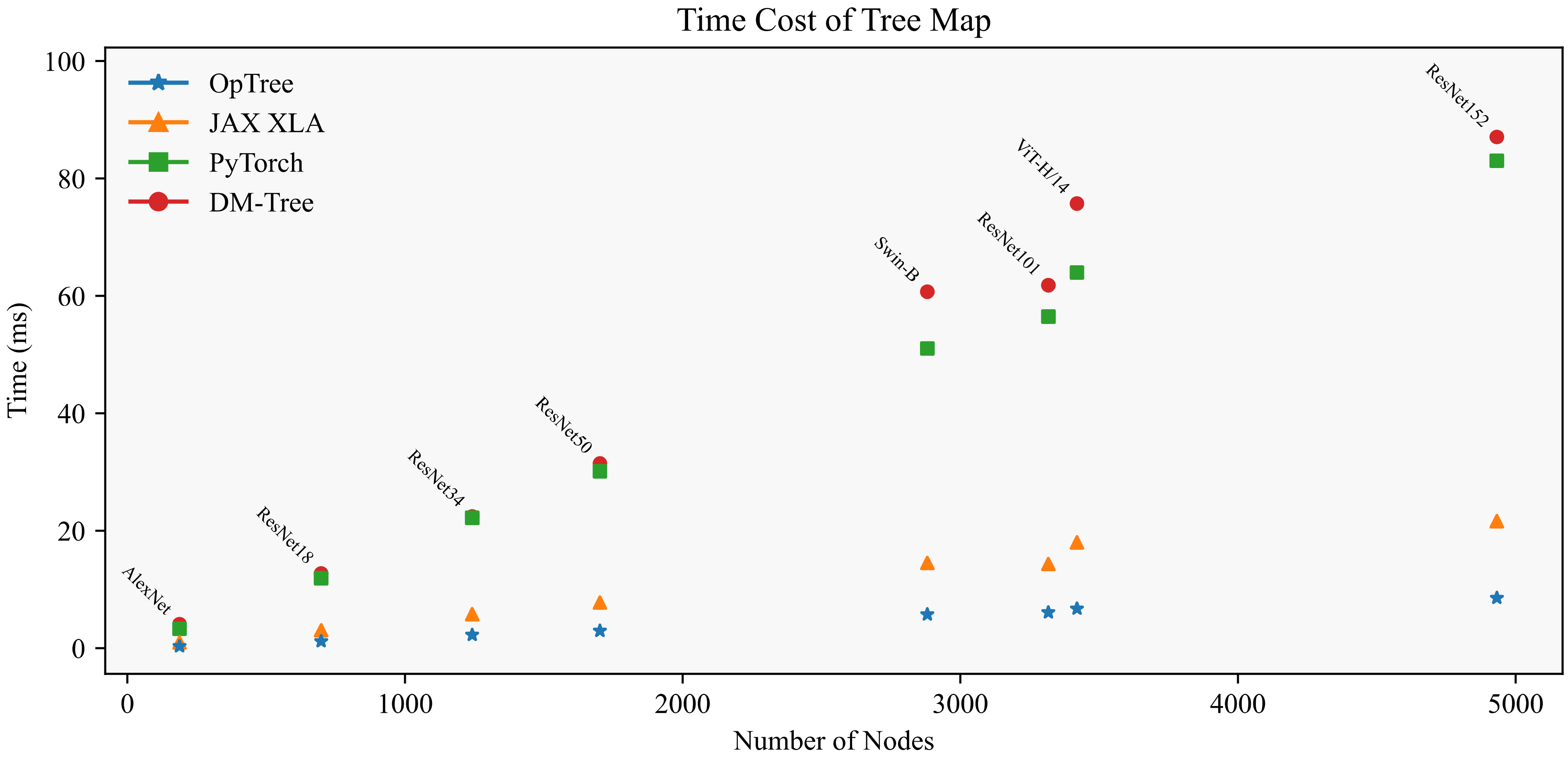
- OpTree: Optimized PyTree Utilities

Table 2: Speedup ratios of tree operations with ResNet models. Here, `O`, `J`, `P`, `D` refer to OpTree, JAX XLA, PyTorch, and DM-Tree, respectively.

| Module Scale | ResNet18 | | | ResNet50 | | | ResNet101 | | | ResNet152 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Speedup Ratio | J / O | P / O | D / O | J / O | P / O | D / O | J / O | P / O | D / O | J / O | P / O | D / O |
| Tree Flatten | 2.80 | 27.31 | 1.49 | 2.63 | 26.52 | 1.40 | 2.46 | 25.18 | 1.38 | 2.56 | 23.25 | 1.28 |
| Tree UnFlatten | 2.68 | 4.47 | 15.89 | 2.56 | 4.16 | 14.51 | 2.55 | 4.32 | 14.86 | 2.68 | 4.51 | 15.70 |
| Tree Map | 2.61 | 10.17 | 10.86 | 2.63 | 10.18 | 10.62 | 2.35 | 9.26 | 10.13 | 2.53 | 9.69 | 10.16 |

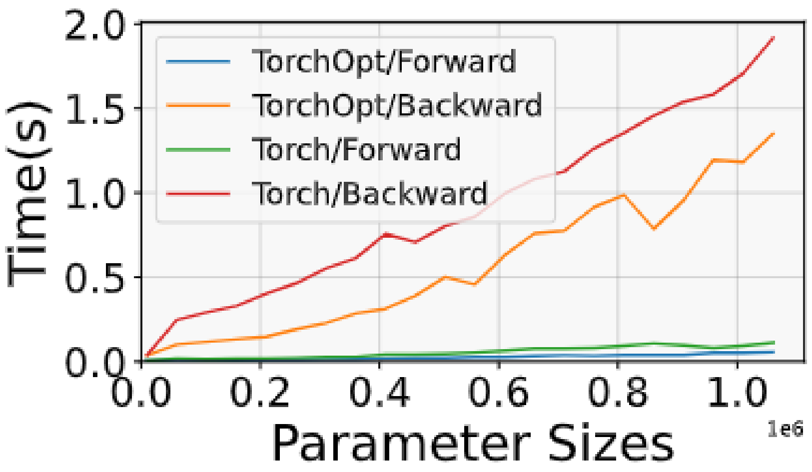
# High-performance and distributed execution runtime

- OpTree: Optimized PyTree Utilities

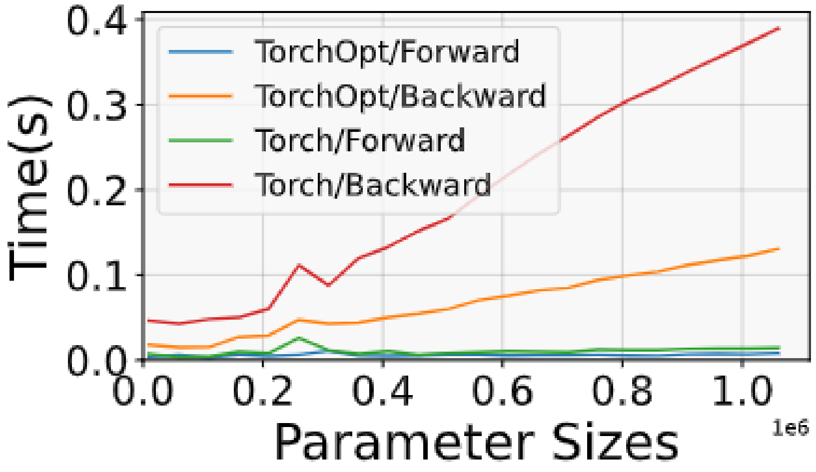


# High-performance and distributed execution runtime

- CPU/GPU-accelerated Optimizers
  - Implement explicit shortcuts (forward/backward) (reduces 30%+ operations)
  - C++ OpenMP & CUDA



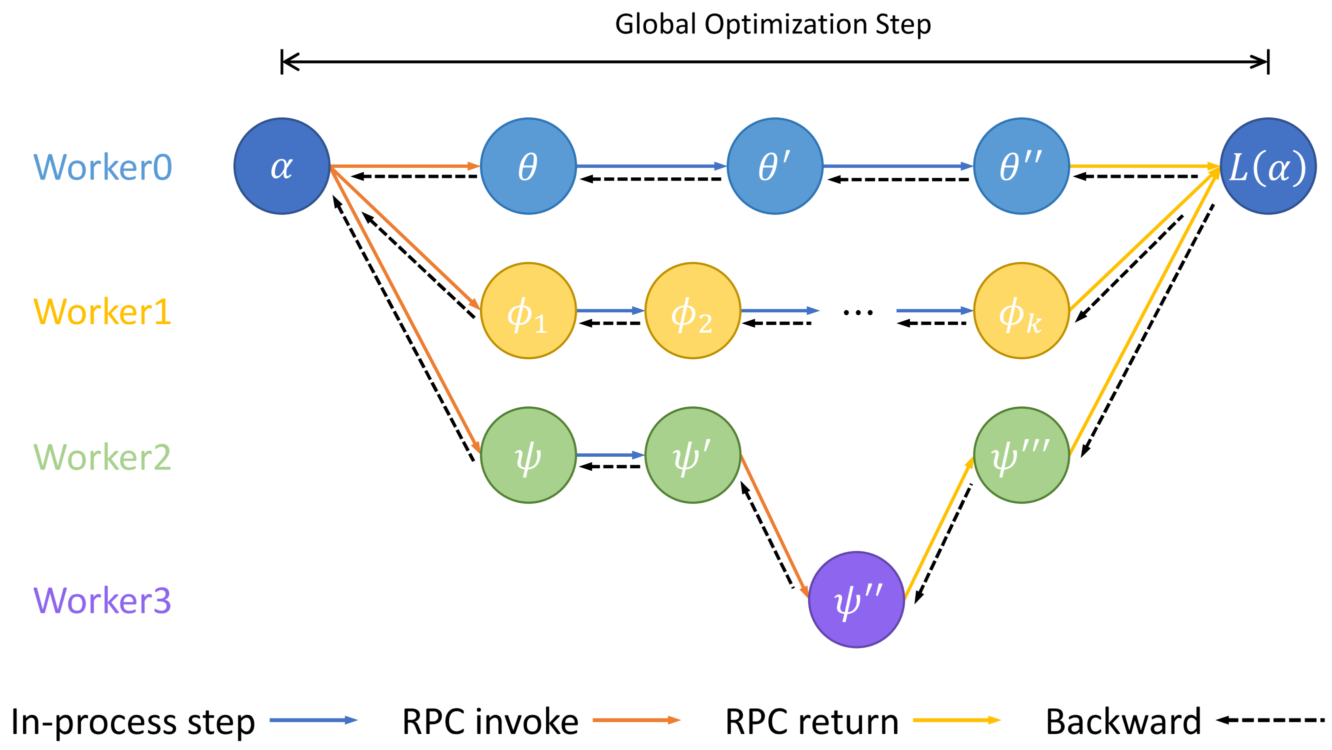

(a) CPU-accelerated optimizer




(b) GPU-accelerated optimizer

# High-performance and distributed execution runtime

- Distributed Training
  - User friendly API (distributed. auto_init_rpc, distributed.backward, )
  - Auto Parallelization



# High-performance and distributed execution runtime

- Distributed Training
  - User friendly API (distributed. auto_init_rpc, distributed.backward, )
  - Auto Parallelization

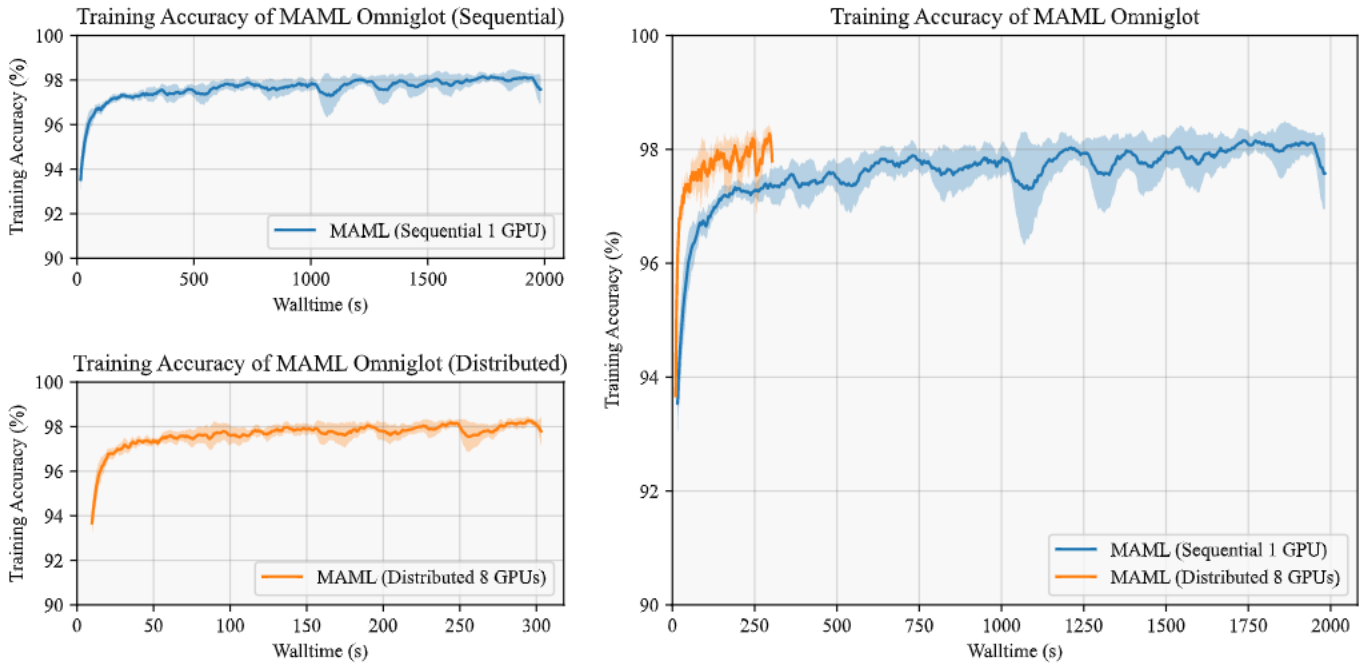
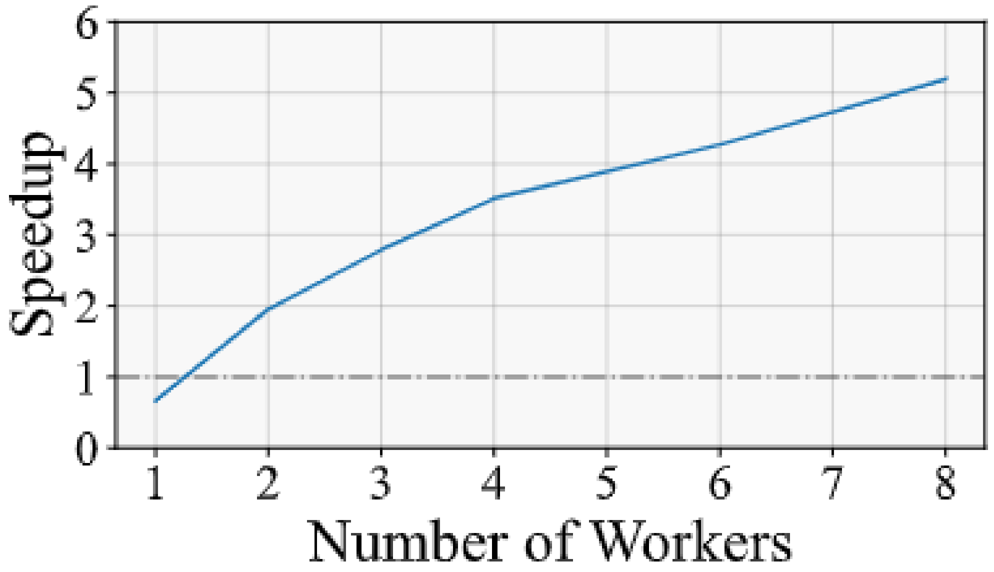



Figure 7: Wall time comparison between sequential training results and distributed training on 8 GPUs for MAML implemented with TorchOpt.

# High-performance and distributed execution runtime

- Distributed Training
  - User friendly API (distributed. auto_init_rpc, distributed.backward, )
  - Auto Parallelization




(c) Distributed Speedup Ratio

# High-performance and distributed execution runtime

- Distributed Training

```python
import torchopt.distributed as todist

def worker_init_fn():
    # set process title, seeding, etc.

@todist.auto_init_rpc(worker_init_fn)
def main():
    ...
    model = Model(...)
    train(model)        # execute on rank 0 only
    save_model(model)   # execute on rank 0 only
```

```python
@todist.rank_zero_only
def save_model(model):
    ...

@todist.rank_zero_only
def train(model):
    model_rref = todist.rpc.RRef(model_rref)
    dataloader = DataLoader(...)
    optimizer = Optimizer(...)

    for batch in dataloader:
        optimizer.zero_grad()
        with todist.autograd.context() as context_id:
            loss = compute_loss(model_rref, batch)
            todist.autograd.backward(context_id, loss)
            optimizer.step()
```

```python
@todist.parallelize(partitioner=todist.batch_partitioner, reducer=todist.mean_reducer)
def compute_loss(model_rref, batch):
    model = model_rref.to_here()
    loss = ...
    return loss
```

# Thank you!


*jieren9806@gmail.com*

*xidong.feng.20@ucl.ac.uk*

*benjaminliu.eecs@gmail.com*

*xuehaipan@pku.edu.cn*


TorchOpt: https://github.com/metaopt/torchopt

OpTree:https://github.com/metaopt/optree