Amanda: Unified Instrumentation Framework for Deep Neural Networks

ASPLOS’24

Yue Guan, Yuxian Qiu, Jingwen Leng, Fan Yang, Shuo Yu, Yunxin Liu, Yu Feng, Yuhao Zhu, Lidong Zhou, Yun Liang, Chen Zhang and Chao Li

Shanghai JiaoTong University, Shanghai Qi Zhi Institute, Microsoft Research, Tsinghua University, Shanghai AI Lab, University of Rochester, Peking University

Background

- Instrumentation in traditional programming analysis

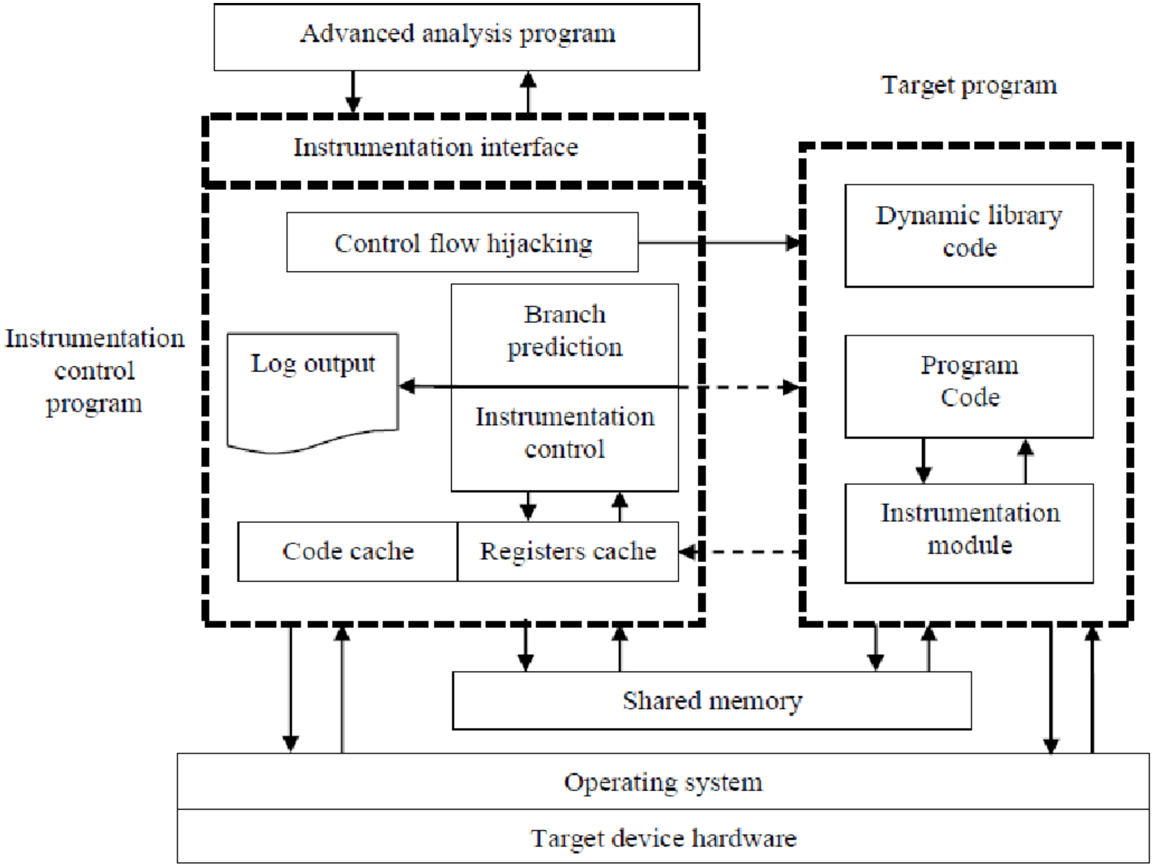
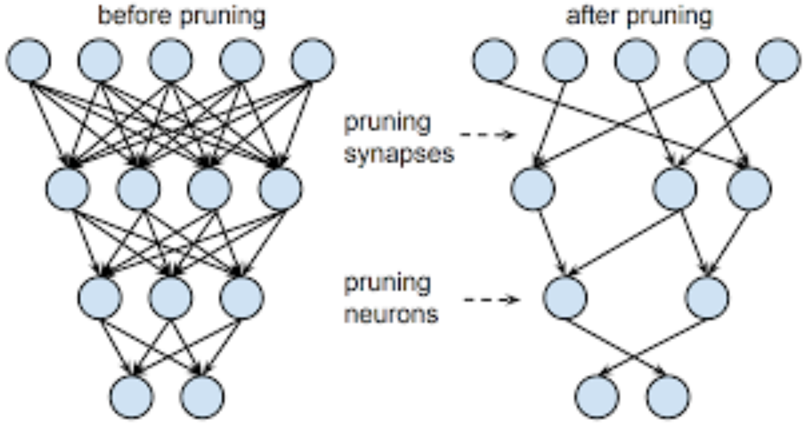
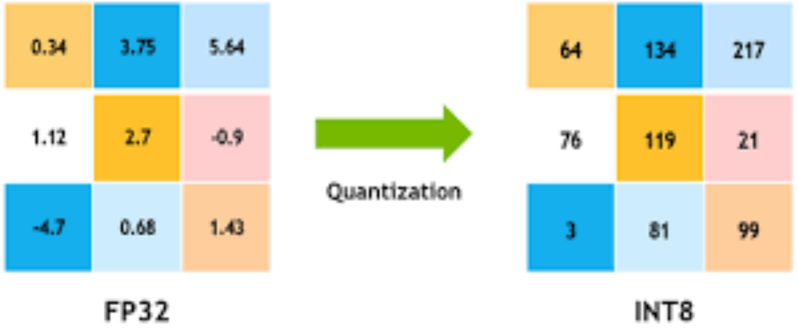


Figure 2. Dynamic binary instrumentation overall framework.

Background and Motivation

- Lots of works focus on analyzing or accelerating DNN inference or training





0.34	3.75	5.64
1.12	2.7	-0.9
-4.7	0.68	1.43

FP32

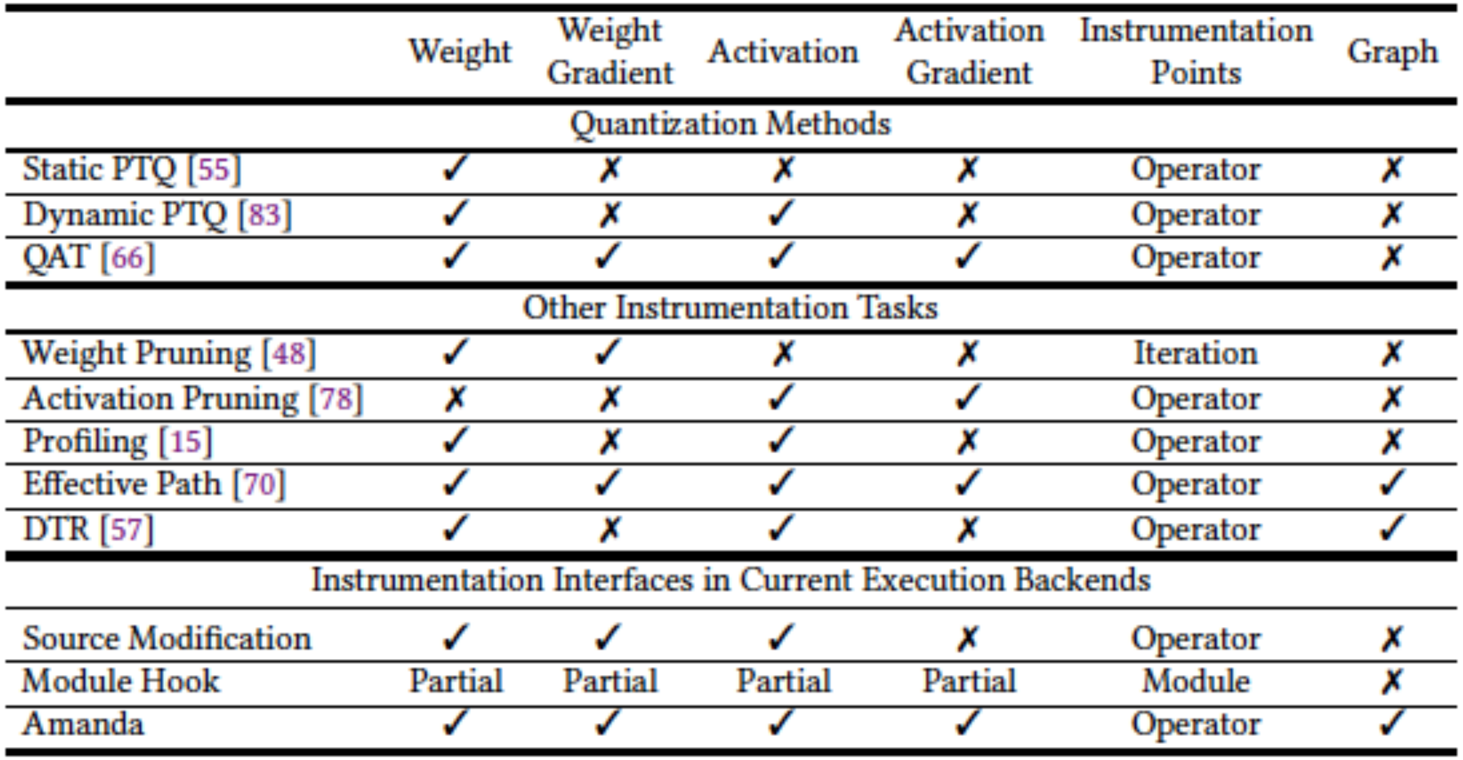
64	134	217
76	119	21
3	81	99

INT8

Alpa

Background and Motivation

- The common requirements of these tasks is to monitor and manipulate the computation process of DNN models.

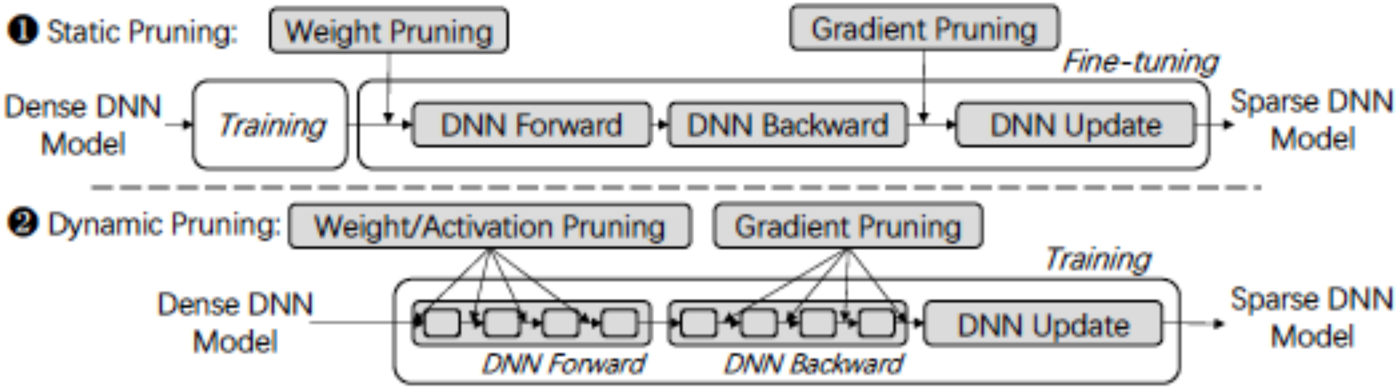


	Weight	Weight Gradient	Activation	Activation Gradient	Instrumentation Points	Graph
Quantization Methods						
Static PTQ [55]	✓	✗	✗	✗	Operator	✗
Dynamic PTQ [83]	✓	✗	✓	✗	Operator	✗
QAT [66]	✓	✓	✓	✓	Operator	✗
Other Instrumentation Tasks						
Weight Pruning [48]	✓	✓	✗	✗	Iteration	✗
Activation Pruning [78]	✗	✗	✓	✓	Operator	✗
Profiling [15]	✓	✗	✓	✗	Operator	✗
Effective Path [70]	✓	✓	✓	✓	Operator	✓
DTR [57]	✓	✗	✓	✗	Operator	✓
Instrumentation Interfaces in Current Execution Backends						
Source Modification	✓	✓	✓	✗	Operator	✗
Module Hook	Partial	Partial	Partial	Partial	Module	✗
Amanda	✓	✓	✓	✓	Operator	✓

Various requirements for implementation

Background and Motivation

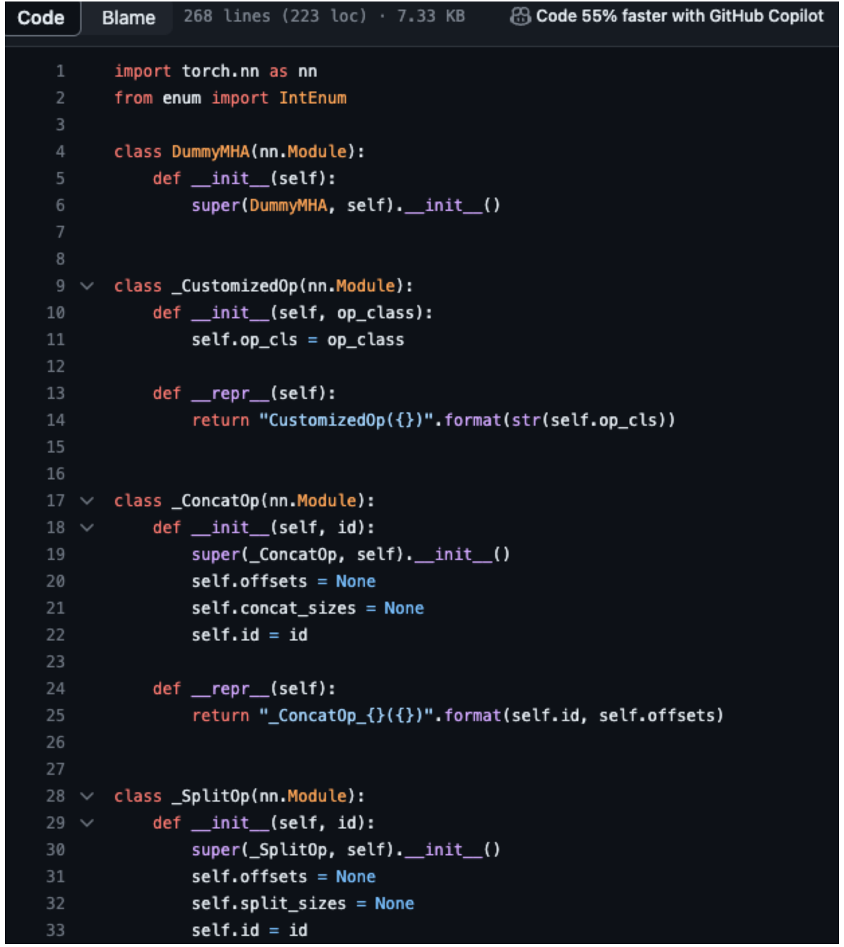
- Problems of existed implementation and backend support
  - Ad-hoc instrumentation points



Background and Motivation

- Problems of existed implementation and backend support
  - Fragmented state representations

Developers often manually wrap every PyTorch module with additional operators.


Code
Blame
268 lines (223 loc) · 7.33 KB
Code 55% faster with GitHub Copilot

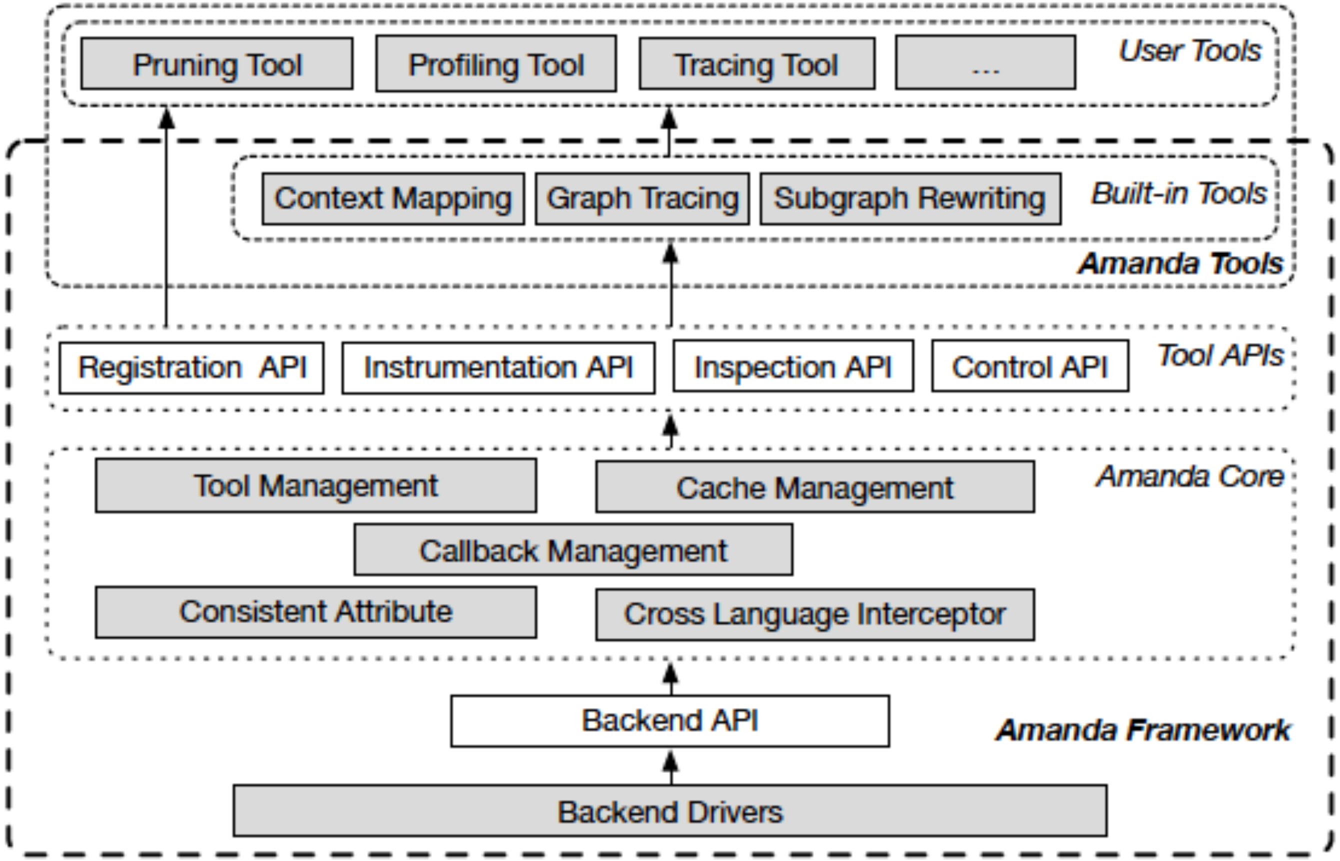
```python
import torch.nn as nn
from enum import IntEnum

class DummyMHA(nn.Module):
    def __init__(self):
        super(DummyMHA, self).__init__()


class _CustomizedOp(nn.Module):
    def __init__(self, op_class):
        self.op_cls = op_class

    def __repr__(self):
        return "CustomizedOp({})".format(str(self.op_cls))


class _ConcatOp(nn.Module):
    def __init__(self, id):
        super(_ConcatOp, self).__init__()
        self.offsets = None
        self.concat_sizes = None
        self.id = id

    def __repr__(self):
        return "_ConcatOp_{}({})".format(self.id, self.offsets)


class _SplitOp(nn.Module):
    def __init__(self, id):
        super(_SplitOp, self).__init__()
        self.offsets = None
        self.split_sizes = None
        self.id = id
```

Background and Motivation

- Problems of existed implementation and backend support
  - Execution modes
    - Graph mode: Cook all, eat one by one.
    - Eager mode: Cook one, eat one.

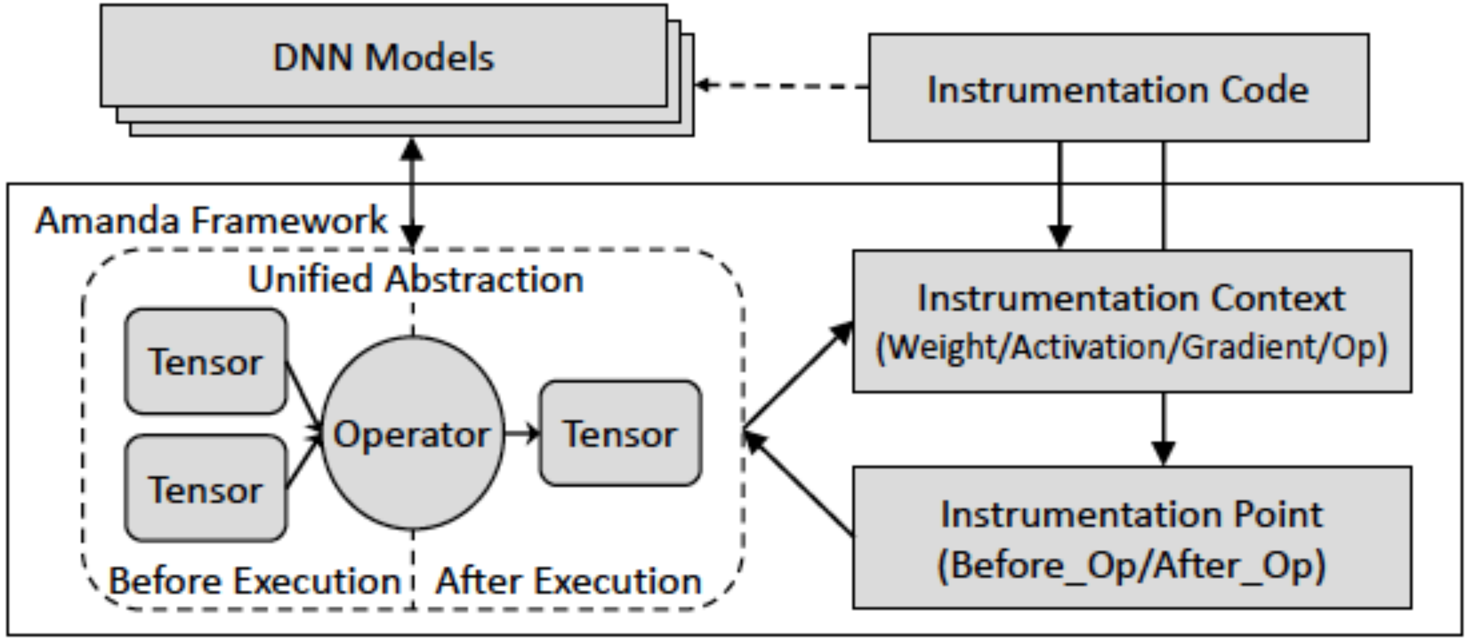
Some tasks need to use graph mode to analyze the DNN which add the extra complexity dimension in implementation.

System Overview



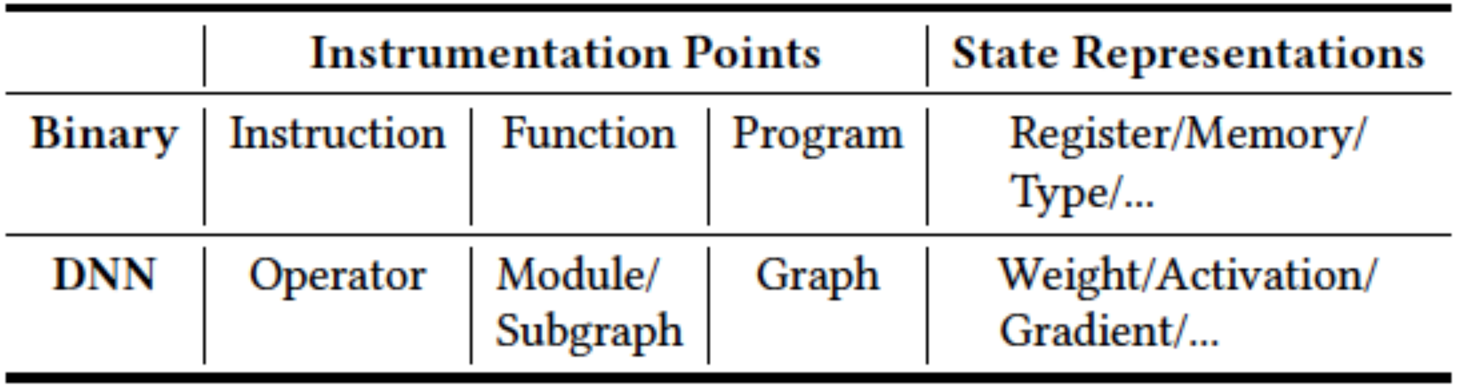
Instrumentation

- Points



Instrumentation

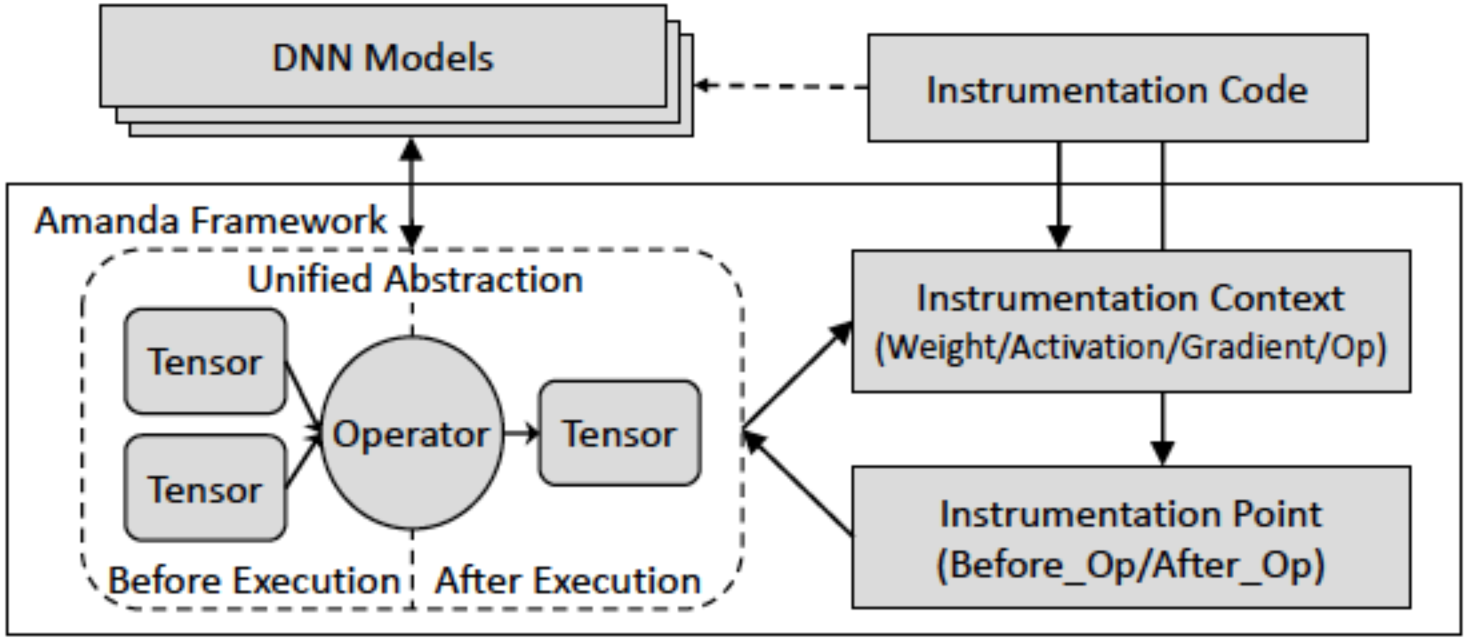
- Compare with traditional concept



	Instrumentation Points			State Representations
Binary	Instruction	Function	Program	Register/Memory/ Type/...
DNN	Operator	Module/ Subgraph	Graph	Weight/Activation/ Gradient/...

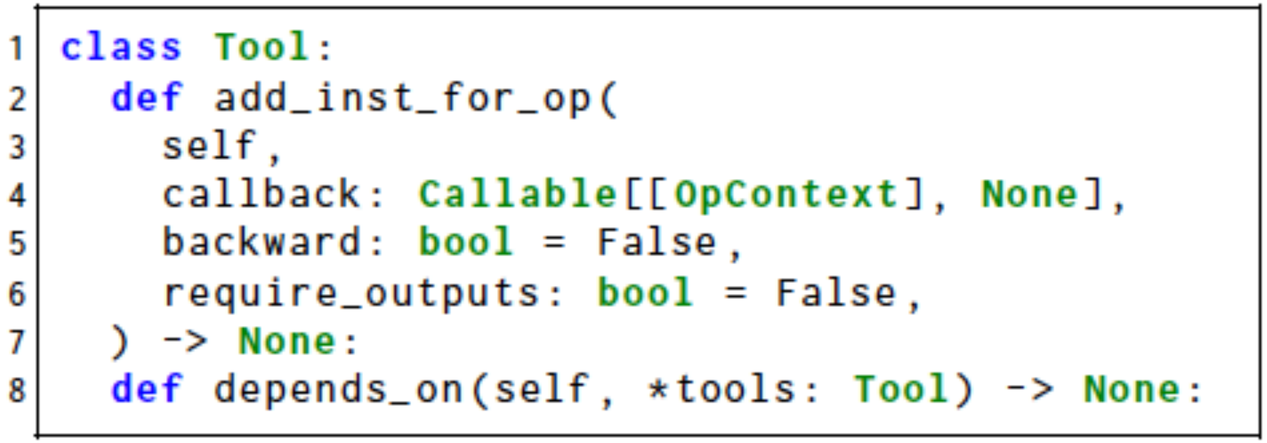
Instrumentation

- Analysis routines and instrumentation routines
  - Analysis routines: Init, check status, locate point and register instrumentations.
  - Instrumentation routines: Modify operators. Execute in runtime states.



Instrumentation

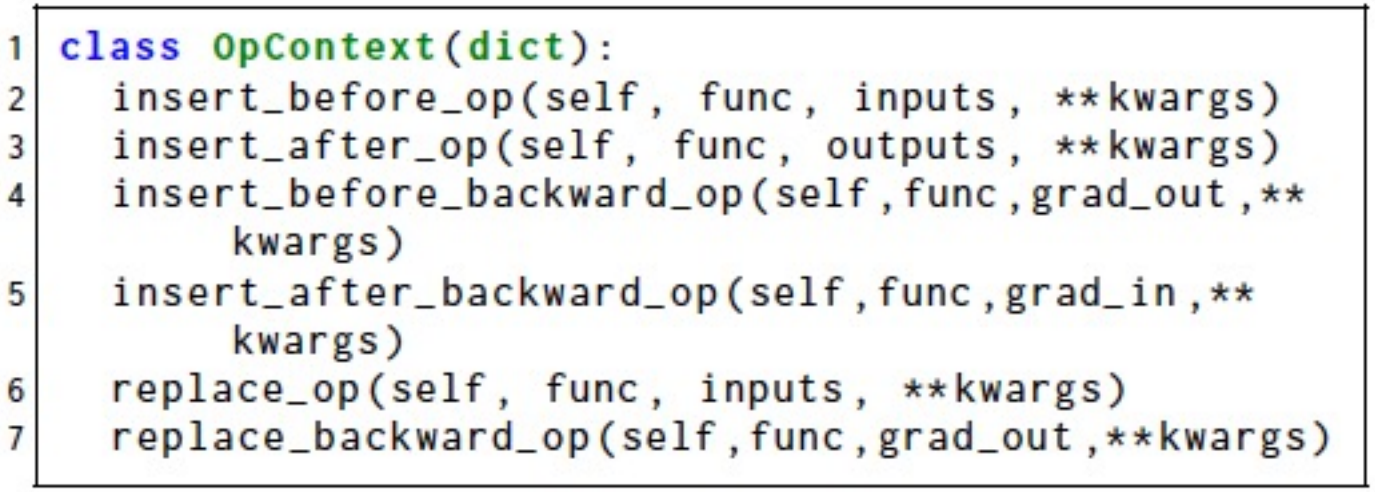
- Tool APIs
  - Registration APIs



```
class Tool:
  def add_inst_for_op(
    self,
    callback: Callable[[OpContext], None],
    backward: bool = False,
    require_outputs: bool = False,
  ) -> None:
  def depends_on(self, *tools: Tool) -> None:
```

Instrumentation

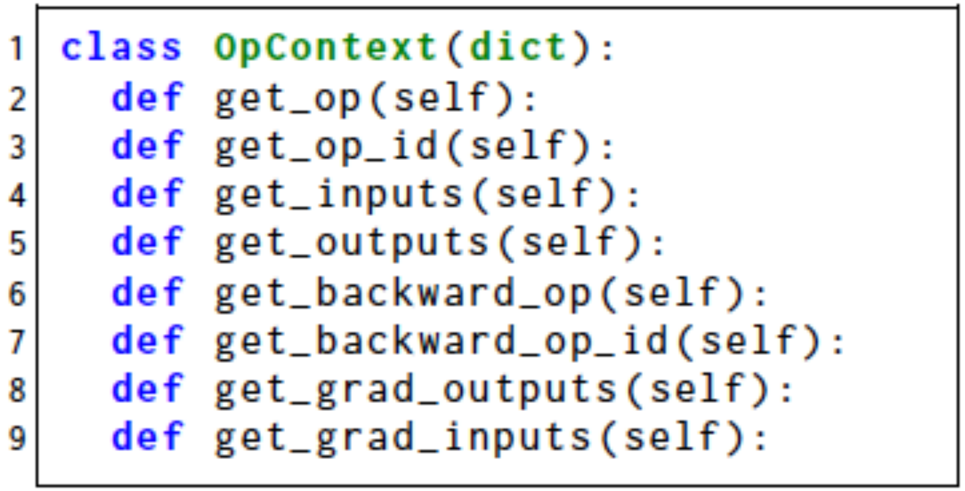
- Tool APIs
  - Instrumentation APIs



```
class OpContext(dict):
  insert_before_op(self, func, inputs, **kwargs)
  insert_after_op(self, func, outputs, **kwargs)
  insert_before_backward_op(self,func,grad_out,**
       kwargs)
  insert_after_backward_op(self,func,grad_in,**
       kwargs)
  replace_op(self, func, inputs, **kwargs)
  replace_backward_op(self,func,grad_out,**kwargs)
```

Instrumentation

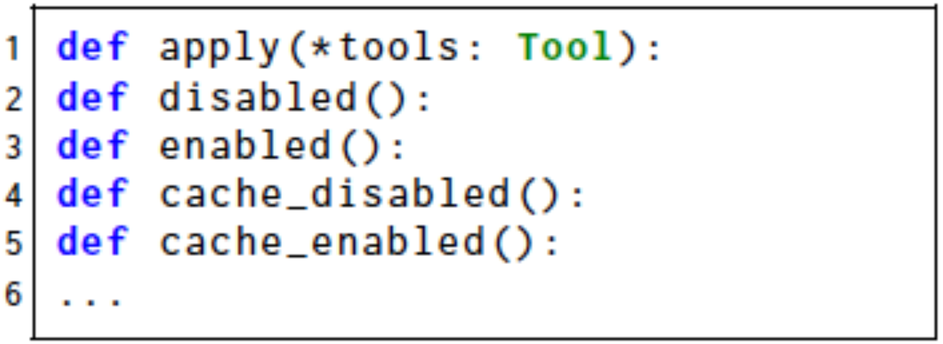
- Tool APIs
    - Inspection APIs



```python
class OpContext(dict):
  def get_op(self):
  def get_op_id(self):
  def get_inputs(self):
  def get_outputs(self):
  def get_backward_op(self):
  def get_backward_op_id(self):
  def get_grad_outputs(self):
  def get_grad_inputs(self):
```

Instrumentation

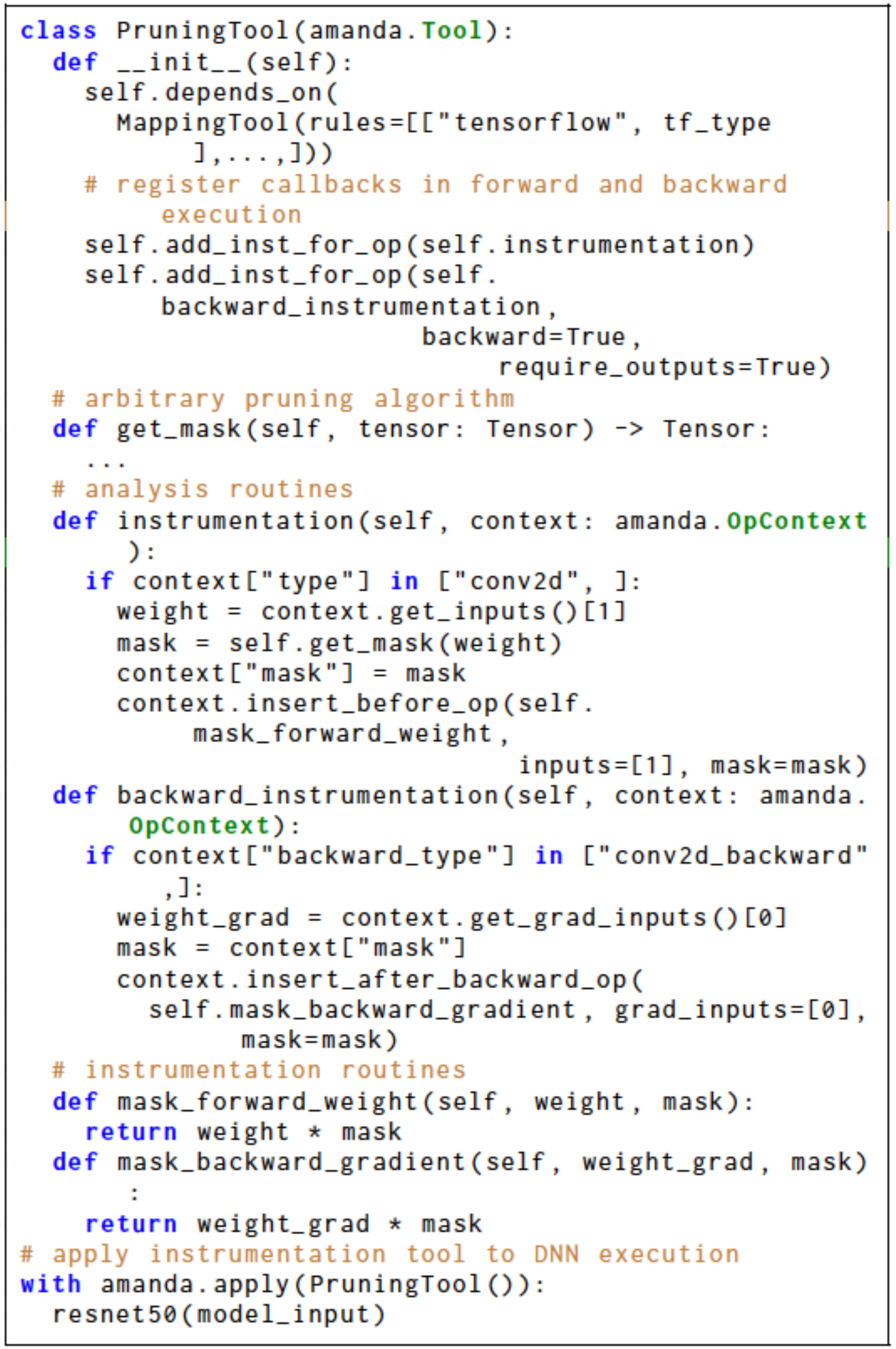
- Tool APIs
  - Control APIs



```
def apply(*tools: Tool):
def disabled():
def enabled():
def cache_disabled():
def cache_enabled():
...
```

Instrumentation

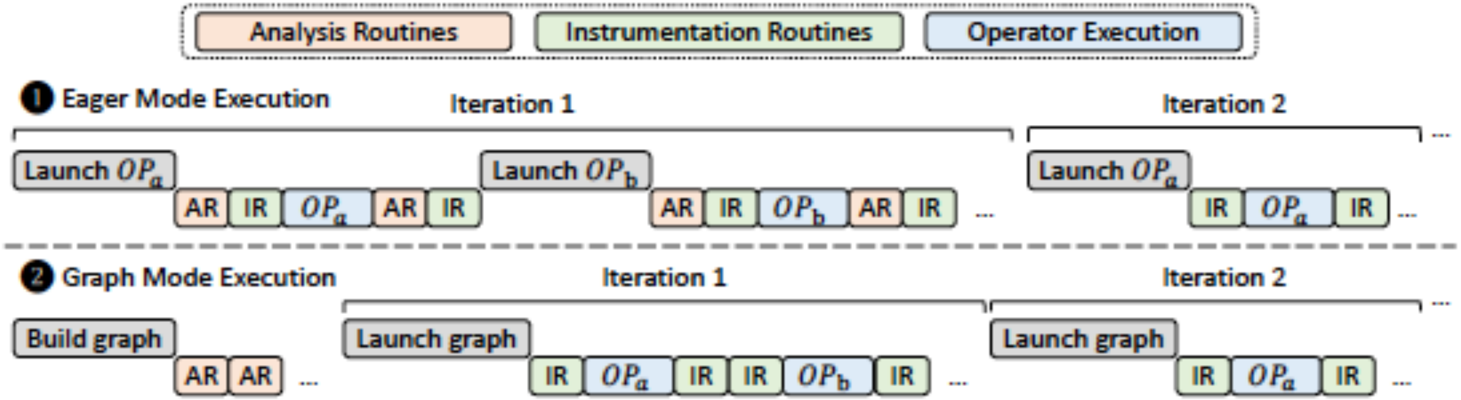
- Pruning case study



```
class PruningTool(amanda.Tool):
  def __init__(self):
    self.depends_on(
      MappingTool(rules=[["tensorflow", tf_type],...,]))
    # register callbacks in forward and backward execution
    self.add_inst_for_op(self.instrumentation)
    self.add_inst_for_op(self.backward_instrumentation,
                         backward=True,
                              require_outputs=True)
  # arbitrary pruning algorithm
  def get_mask(self, tensor: Tensor) -> Tensor:
    ...
  # analysis routines
  def instrumentation(self, context: amanda.OpContext):
    if context["type"] in ["conv2d", ]:
      weight = context.get_inputs()[1]
      mask = self.get_mask(weight)
      context["mask"] = mask
      context.insert_before_op(self.mask_forward_weight,
                               inputs=[1], mask=mask)
  def backward_instrumentation(self, context: amanda.OpContext):
    if context["backward_type"] in ["conv2d_backward",]:
      weight_grad = context.get_grad_inputs()[0]
      mask = context["mask"]
      context.insert_after_backward_op(
        self.mask_backward_gradient, grad_inputs=[0], mask=mask)
  # instrumentation routines
  def mask_forward_weight(self, weight, mask):
    return weight * mask
  def mask_backward_gradient(self, weight_grad, mask):
    return weight_grad * mask
# apply instrumentation tool to DNN execution
with amanda.apply(PruningTool()):
  resnet50(model_input)
```

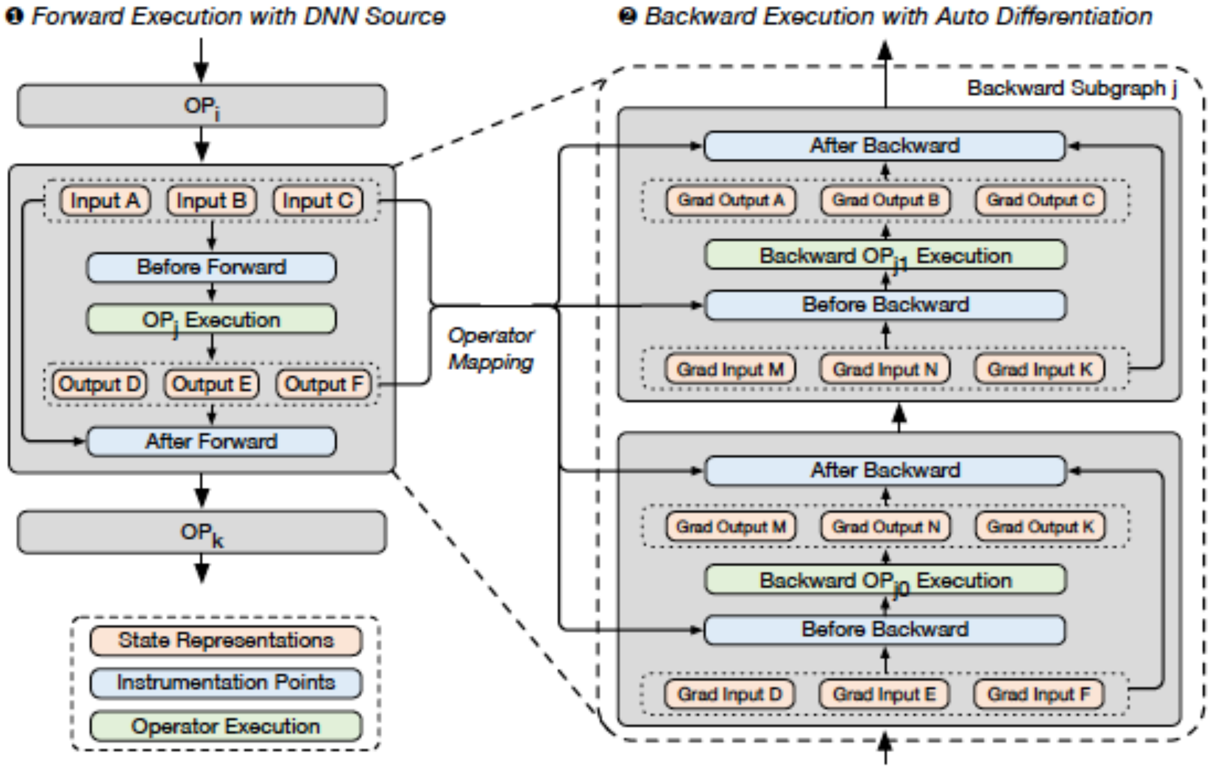
Amanda Core

- Harmonizing instrumentation semantics on different execution modes



Amanda Core

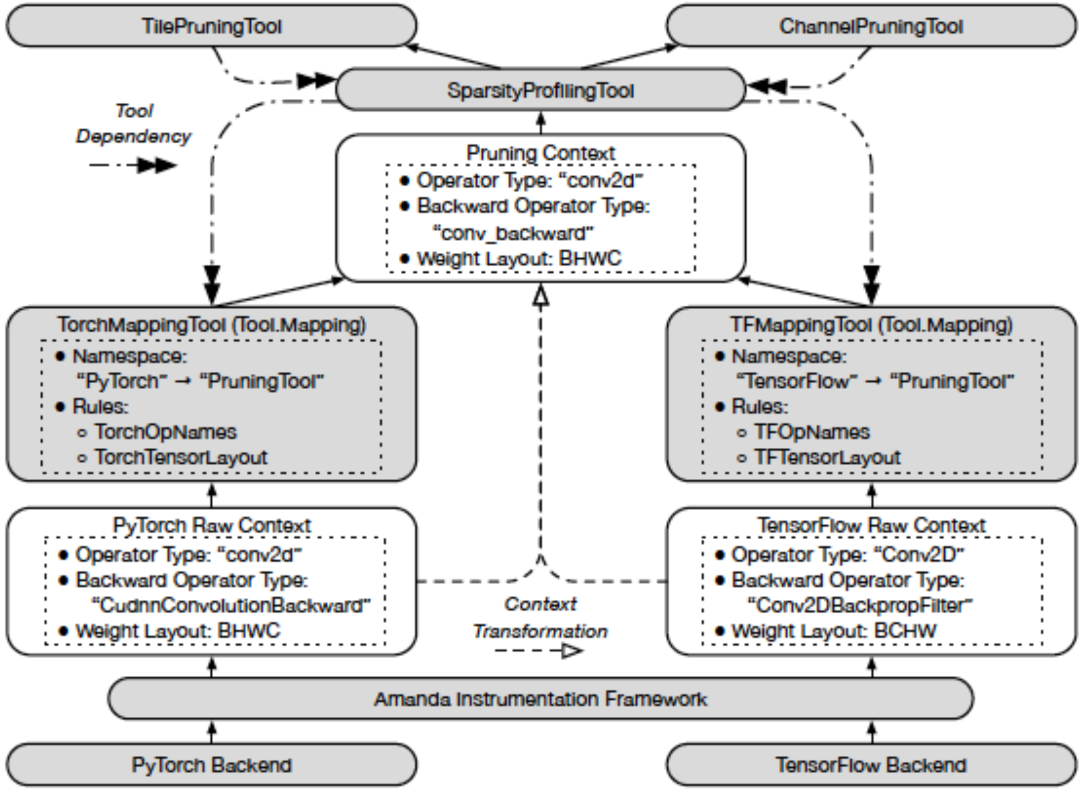
- Addressing AD mechanism



- Original + AD program
- Enable or disable control

Amanda Core

- Composable tools and context transformation

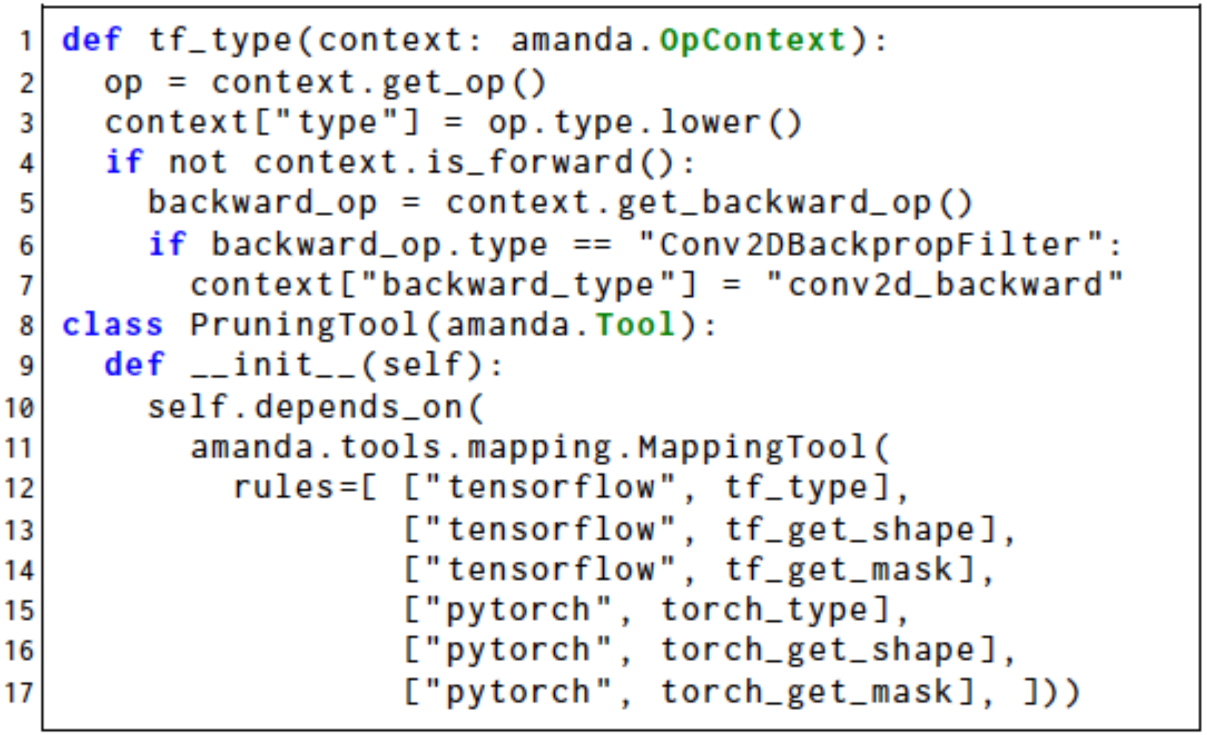


TilePruningTool
ChannelPruningTool
SparsityProfilingTool
Tool Dependency
Pruning Context
• Operator Type: "conv2d"
• Backward Operator Type: "conv_backward"
• Weight Layout: BHWC
TorchMappingTool (Tool.Mapping)
• Namespace: "PyTorch" → "PruningTool"
• Rules:
∘ TorchOpNames
∘ TorchTensorLayout
TFMappingTool (Tool.Mapping)
• Namespace: "TensorFlow" → "PruningTool"
• Rules:
∘ TFOpNames
∘ TFTensorLayout
PyTorch Raw Context
• Operator Type: "conv2d"
• Backward Operator Type: "CudnnConvolutionBackward"
• Weight Layout: BHWC
Context Transformation
TensorFlow Raw Context
• Operator Type: "Conv2D"
• Backward Operator Type: "Conv2DBackpropFilter"
• Weight Layout: BCHW
Amanda Instrumentation Framework
PyTorch Backend
TensorFlow Backend

- Dependency registration
- Resolving the dependency graph of instrumentation tools during initialization and detecting loop dependencies.

Amanda Core

- Composable tools and context transformation



```python
def tf_type(context: amanda.OpContext):
  op = context.get_op()
  context["type"] = op.type.lower()
  if not context.is_forward():
    backward_op = context.get_backward_op()
    if backward_op.type == "Conv2DBackpropFilter":
      context["backward_type"] = "conv2d_backward"
class PruningTool(amanda.Tool):
  def __init__(self):
    self.depends_on(
      amanda.tools.mapping.MappingTool(
        rules=[ ["tensorflow", tf_type],
                ["tensorflow", tf_get_shape],
                ["tensorflow", tf_get_mask],
                ["pytorch", torch_type],
                ["pytorch", torch_get_shape],
                ["pytorch", torch_get_mask], ]))
```

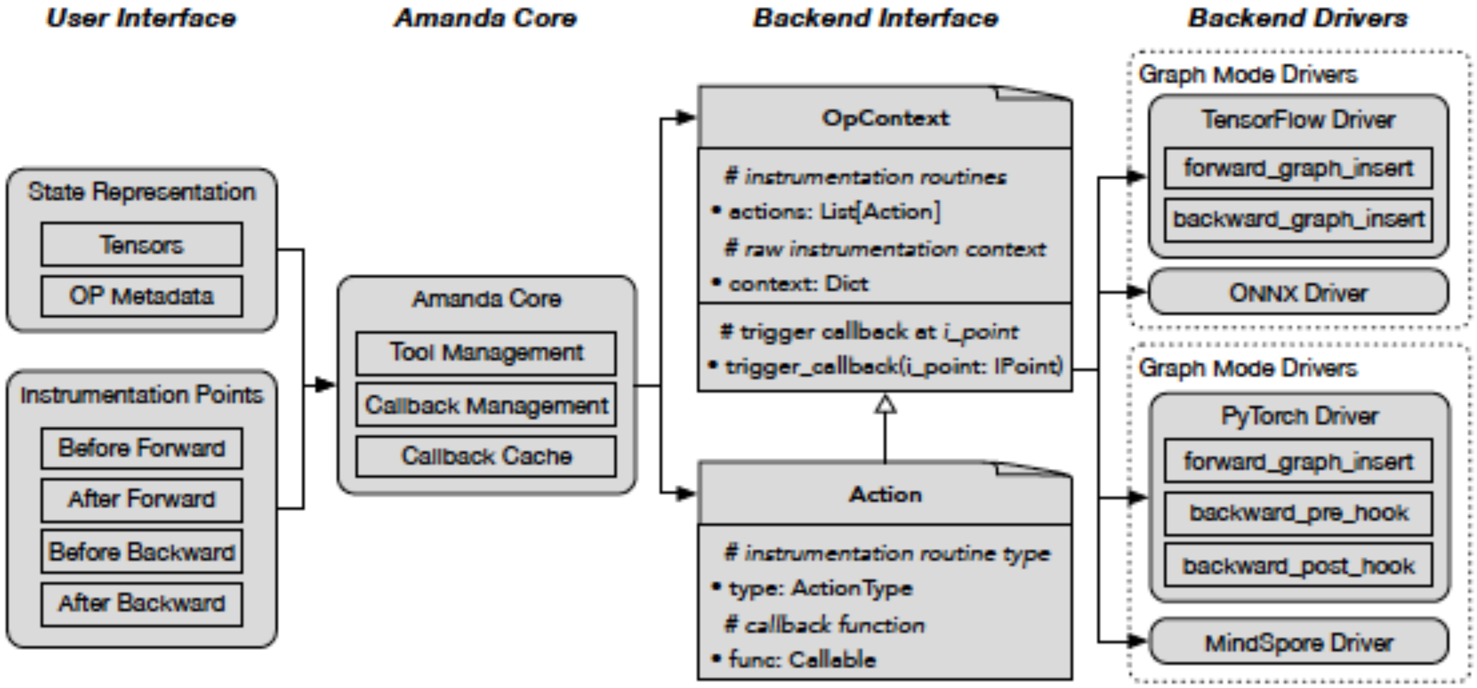
- Mapping tool makes tools are portable

Amanda Core

- Minimizing instrumentation overhead via caching
- Addressing the language disparity

Amanda Backend

- Backend interface

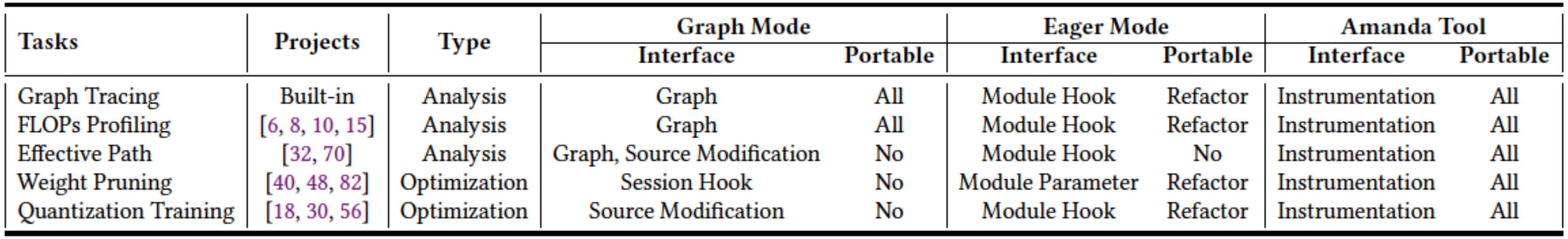


- Monkey-patching
- Traversing mapping

- Graph switching
- Store original and instrumented graphs

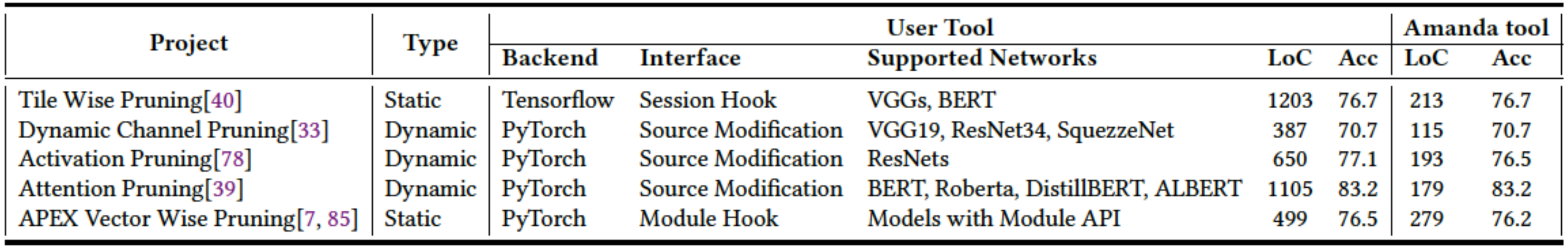
Results

- Tasks



Tasks	Projects	Type	Graph Mode		Eager Mode		Amanda Tool	
			Interface	Portable	Interface	Portable	Interface	Portable
Graph Tracing	Built-in	Analysis	Graph	All	Module Hook	Refactor	Instrumentation	All
FLOPs Profiling	[6, 8, 10, 15]	Analysis	Graph	All	Module Hook	Refactor	Instrumentation	All
Effective Path	[32, 70]	Analysis	Graph, Source Modification	No	Module Hook	No	Instrumentation	All
Weight Pruning	[40, 48, 82]	Optimization	Session Hook	No	Module Parameter	Refactor	Instrumentation	All
Quantization Training	[18, 30, 56]	Optimization	Source Modification	No	Module Hook	Refactor	Instrumentation	All

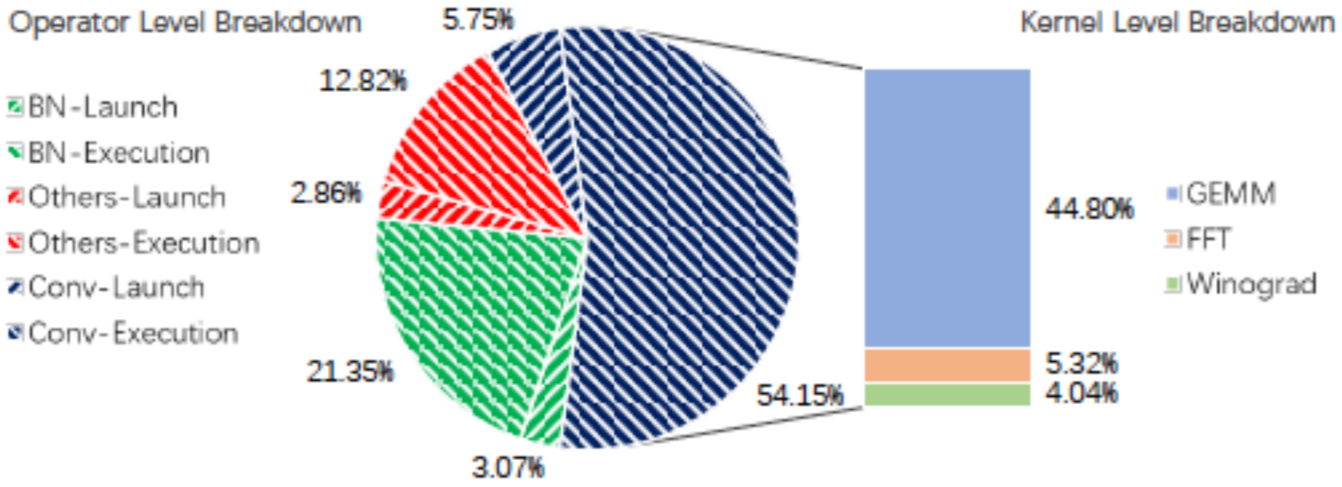
- Generality



Project	Type	User Tool					Amanda tool	
		Backend	Interface	Supported Networks	LoC	Acc	LoC	Acc
Tile Wise Pruning[40]	Static	Tensorflow	Session Hook	VGGs, BERT	1203	76.7	213	76.7
Dynamic Channel Pruning[33]	Dynamic	PyTorch	Source Modification	VGG19, ResNet34, SquzzeNet	387	70.7	115	70.7
Activation Pruning[78]	Dynamic	PyTorch	Source Modification	ResNets	650	77.1	193	76.5
Attention Pruning[39]	Dynamic	PyTorch	Source Modification	BERT, Roberta, DistillBERT, ALBERT	1105	83.2	179	83.2
APEX Vector Wise Pruning[7, 85]	Static	PyTorch	Module Hook	Models with Module API	499	76.5	279	76.2

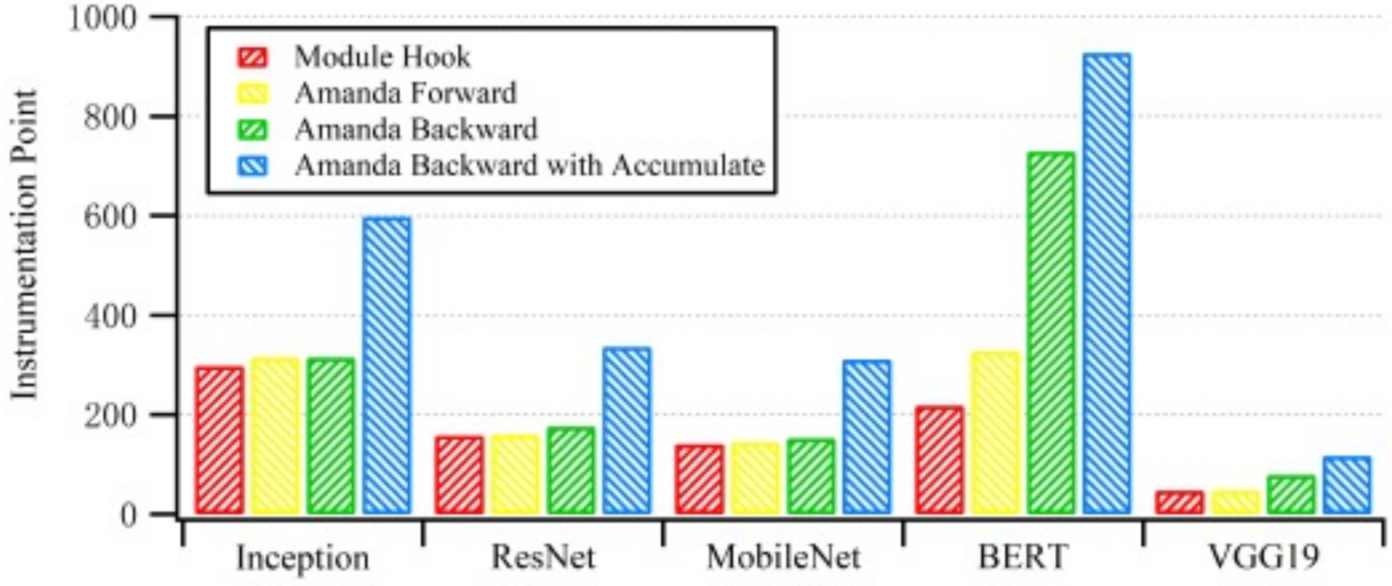
Results

- Integrated with GPU analysis API



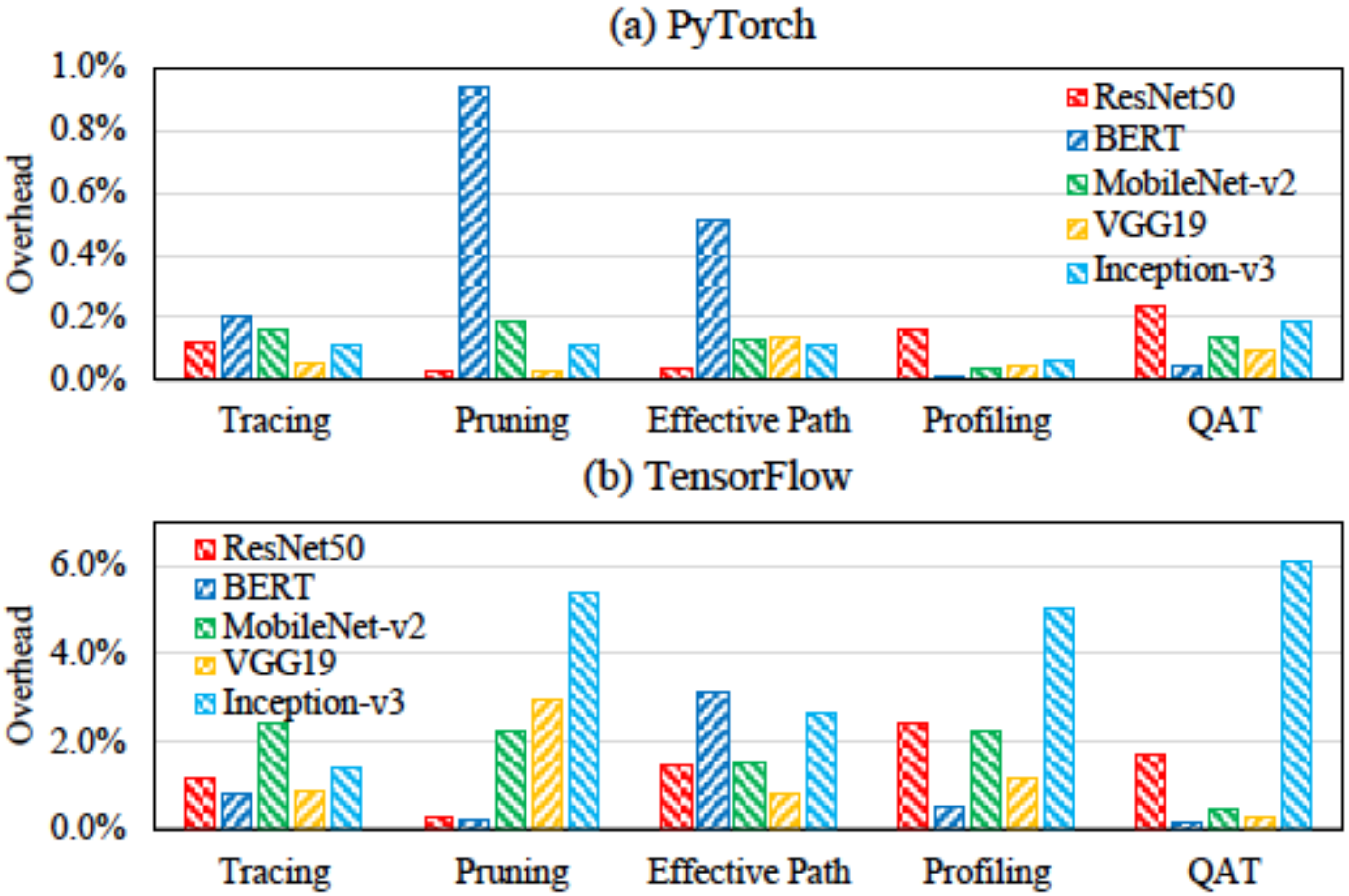
Results

- Instrumentation point coverage



Results

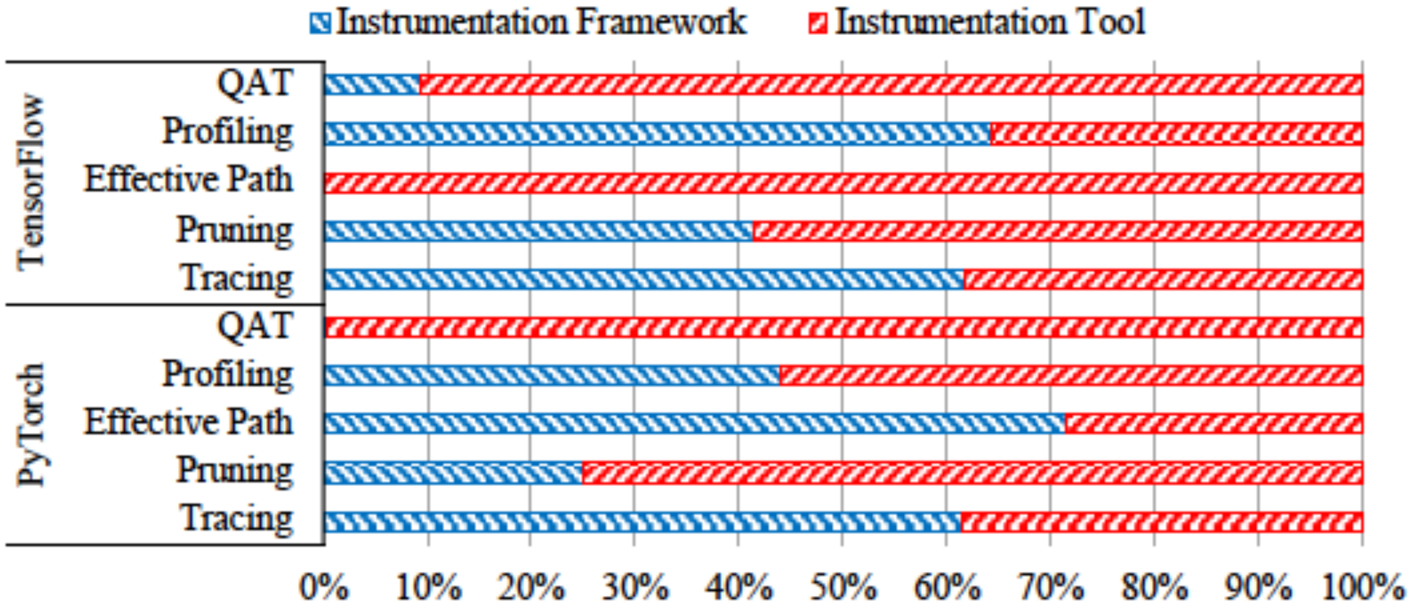
- Overhead


(a) PyTorch

(b) TensorFlow

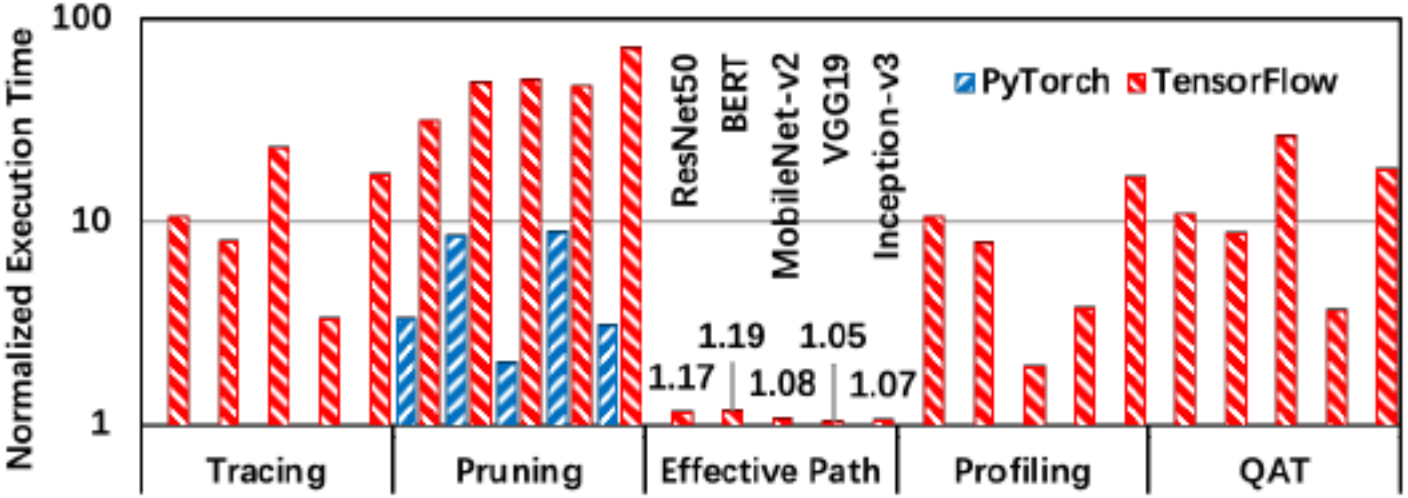
Results

- Execution time breakdown



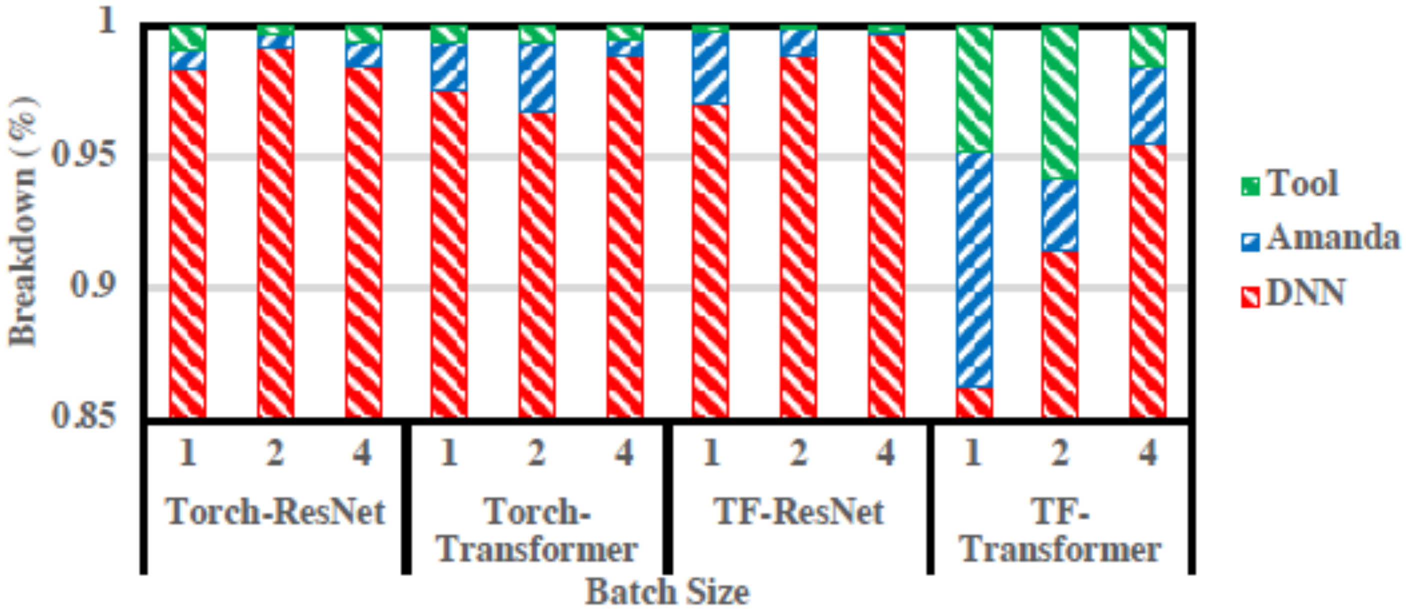
Results

- Memory footprint saved by cache mechanism



Results

- Memory footprint breakdown



Thoughts

- Expect the source code to be released.
- System work full of optimization.
- Driver may be hard to write.

Thank You!

Nov 13, 2023

Presented by Mengyang Liu