# 采用JIT技术解决AI大模型动态图性能问题的创新实践

部门：语言虚拟机实验室
作者：吴江铭
日期：2024/4/29

Security Level:

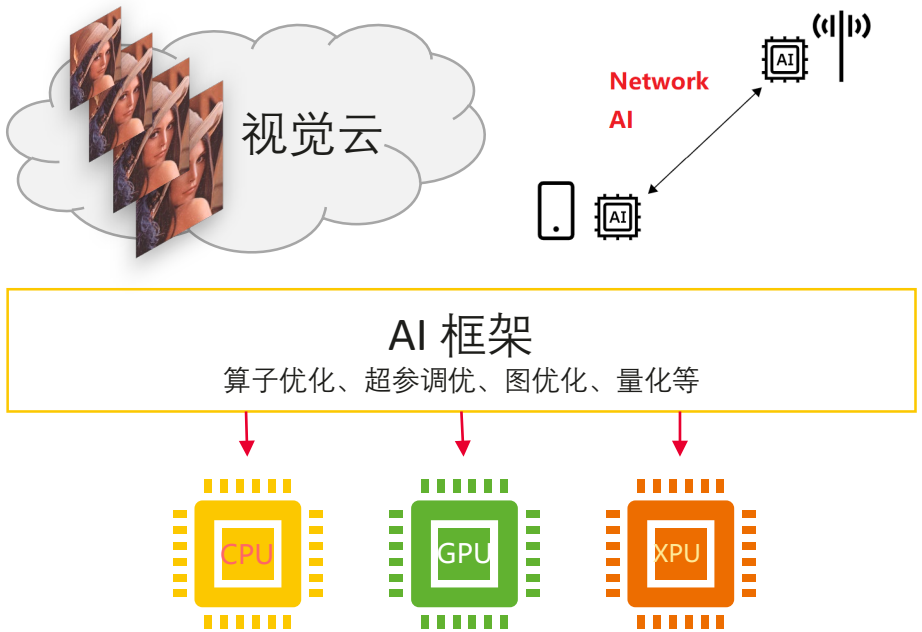
HUAWEI

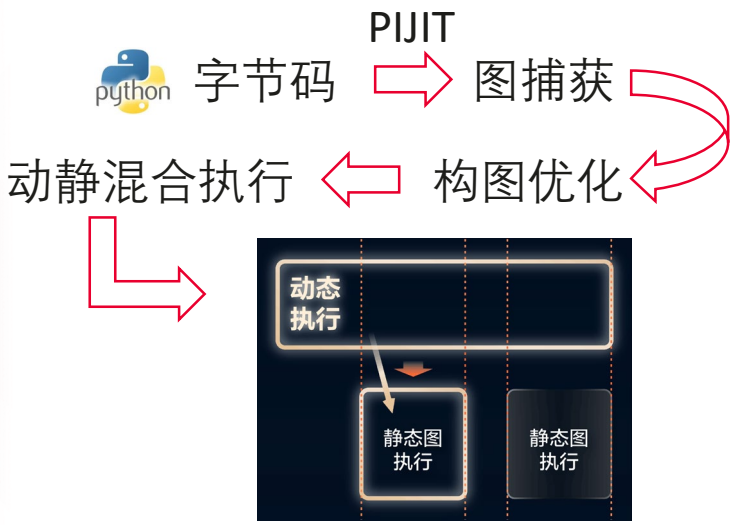
# 目录


1. 个人介绍
2. AI框架编译模型的一些困境
3. 传统JIT & AOT
4. AI框架JIT发展史
5. AI框架JIT方式
6. JIT技术解决AI大模型动态图性能所面临的挑战和方案

# 个人介绍

- 吴江铭，来自华为语言虚拟机实验室
- 熟悉AI领域，擅长CPU、GPU、NPU等体系架构上的算子优化、超参调优、图优化、量化等工作。曾任职于高通、intel等公司，作为AI Tech Leader，主导过NetworkAI、视觉云平台深度学习框架、多种AI加速卡项目。
- 当前主要负责PIJIT项目，采用JIT技术，基于Python字节码，对AI的神经网络脚本，进行图捕获、构图、图优化。




# AI框架编译模型的一些困境

AI模型的编译器不同于传统的编译器，需要考虑高级语言/DSL、面向神经网络的优化（并行、微分、动态）等，比如：

- Python语法的静态化：（易用性）

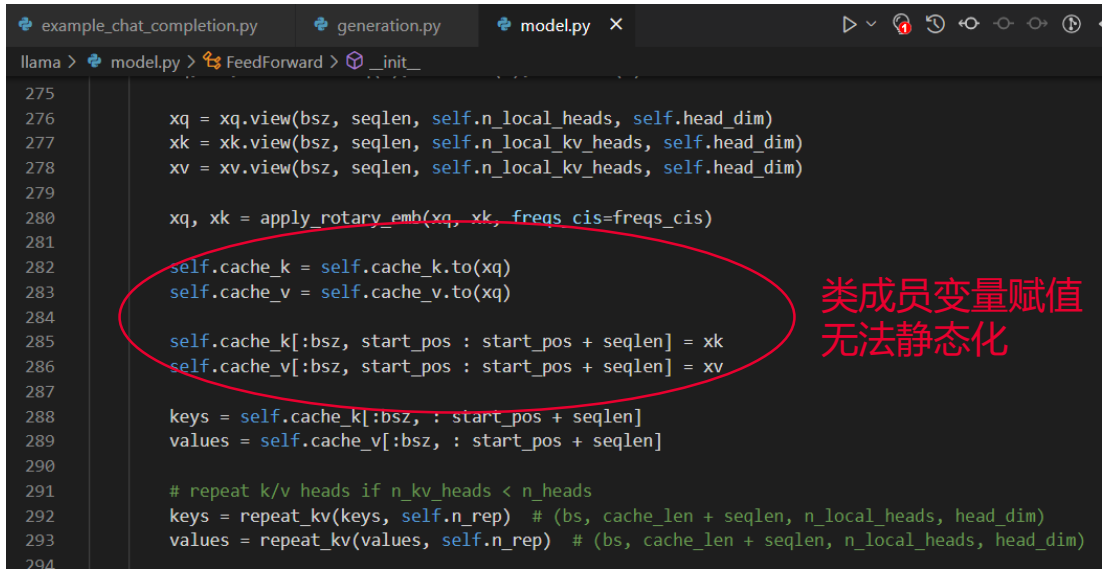
AI编译器需要将Python这种动态类型语言转换为静态的中间表示，但是有些灵活的语法几乎不支持：

✓ try... except...else...finally

✓ with

✓ C Native Function等

- SideEffect副作用: 静态图无法完成某些变量操作的闭包




- 动态特性（性能）

✓ 由于Python是门动态语言，函数类型的动态不确定性影响了静态化的编译性能。

```
>>> def add_func(x, y):
...     return x + y
>>> a=torch.tensor([[1, 1], [1, 1]])
>>> b=torch.tensor([[2, 2], [2, 2]])
>>> print(add_func(a,b))
 tensor([[3,3],[3,3]])
>>> a=torch.tensor([[1, 1]])
>>> b=torch.tensor([[2, 2]])
>>> print(add_func(a,b))
  tensor([[3,3]])
```

✓ Tensor的Shape依赖于具体的运算，无法提前通过计算得出。具体来说分两种情况：算子输入是动态Shape和算子输出是动态Shape。

```
>>> x = torch.randn(3, 4)
>>> x
 tensor([[ 0.3552, -2.3825, -0.8297, 0.3477], [-1.2035, 1.2252, 0.5002, 0.6248], [ 0.1307, -2.0608, 0.1244, 2.0139]])
>>> mask = x.ge(0.5)
>>> mask
 tensor([[False, False, False, False], [False, True, True, True], [False, False, False, True]])
>>> torch.masked_select(x, mask)
 tensor([ 1.2252, 0.5002, 0.6248, 2.0139])
```

# 传统JIT & AOT

- AOT & JIT

AOT – Ahead of Time 提前编译

✓ **程序运行前**将代码编译成机器码

✓ 优点：

- ➢ 避免运行时编译性能消耗
- ➢ 加快程序启动
- ➢ 不占用额外内存存储编译信息
- ➢ 安全性更好，机器码难以翻译

✓ 缺点：

- ➢ 运行时信息缺失，无法运行时优化
- ➢ 跨模块优化限制

JIT – Just in Time 即时编译

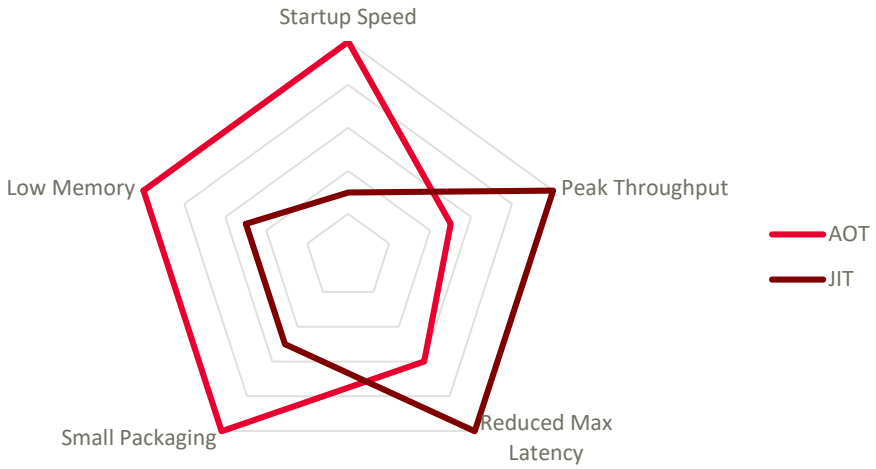
✓ **程序运行时**将代码编译成机器码

✓ 优点：

- ➢ 运行时优化，例如PGO
- ➢ 动态特性支持，例如内联
- ➢ 动态调整代码

✓ 缺点：

- ➢ 启动时间长
- ➢ 实时编译导致卡顿，编译优化和运行时需平衡
- ➢ 更多内存用于保存编译结果




由于动态类型运行前无法确定，
动态语言更适合于JIT的优化，
反而AOT优化不太明显

# AI框架JIT发展史

JAX通过jit，将trace结果转换成jaxpr，进行后端XLA优化。

2015年，Google发布Tensorflow 0.1版本。

2018年，Google发布JAX（Just Aftermath of eXperiments），支持自动微分（grad）、自动分布式（vmap）、即时编译（jit）等。

2017年，Google在Tensorflow中发布XLA（Accelerated Linear Algebra）。

2018年，Tensorflow中发布jit_scope支持局部的JIT优化，在tf1.5版本中以实验方式出现。

2019年，tensorflow 2.0发布，开始支持EagerMode，并使能了函数的jit（tf.function）。从此摆脱Session。

Tf.function通过jit实现了多态（polymorphic)和追踪（trace）

2021年，OpenAI发布Triton1.0，简化了AI算子Kernel的开发。

Triton自定义了AI算子的原语（DSL）并通过JIT在运行时进行编译调优。

1991年

2015年

2017年

2018年

2019年

2021年

2022年

1991年，Python由荷兰国家数学与计算机科学研究中心的吉多·范罗苏姆发布。

2017年，Pytorch第一个版本发布。

2018年，Pytorch发布1.0版本，使能了torch.script和torch.trace。

| UX | Pros | Cons |
| --- | --- | --- |
| **out-of-box (best UX)**<br>*recapture & play system*<br>Lazy Tensor<br>TorchDynamo | • Flip-switch<br>• Always-succeed capture<br>• Sound (re)play | • Possible partial, multi-graphs<br>• Python fallback needed |
| **human-in-the-loop**<br>*capture & replay system*<br>`torch.jit.script`<br>`torch.fx`<br>AOTAutograd | • User-directed, i.e., customizable<br>• Whole-graph if capture succeeds<br>• Python fallback not needed | • Capture may fail (i.e., bes effort)<br>• User-directed, i.e., harde to use |
| **best-effort**<br>*capture & replay system*<br>`torch.jit.trace` | • Flip-switch<br>• Always-succeed whole graph capture<br>• Python-fallback not needed | • Possible unsound replay |

2022年，Pytorch2.0发布新特性TorchDynamo。

Pytorch的三种jit方式已完备：script、trace、dynamo。

采用JIT技术在保有原来的易用性的条件下，提升性能，甚至于通过定义DSL，进一步提升易用性。

# AI框架的JIT方式

- 跟踪型（trace）

通过框架定义的算子实现，跟踪每个算子的执行，记录下顺序，在最后构图优化，并执行。

优点：

✓ 静态图编译一定能够成功。

✓ 图的执行不会fallback回Python。

缺点：

✓ 忽视了未捕获的调用，比如Python，分支控制流，副作用（SideEffect)。

✓ 可能运行结果和预期不符，比如分支流错误。

```
y = 0

# @jit   # Different behavior with jit
def impure_func(x):
  print("Inside:", y)
  return x + y

for y in range(3):
  print("Result:", impure_func(y))
```

```
Inside: 0
Result: 0
Result: 1
Result: 2
```

- 案例

✓ Jax/tf.function

Jax通过自定义的jaxpr，在程序运行过程中，逐个调用Python原语tracer，生成jaxpr（trace_to_jaxpr_final），再转为mlir，交由XLA去优化并生成生成静态图。

```
class EvalTrace(Trace):

  def process_call(self, primitive, f, tracers, params):
    if config.debug_key_reuse.value:
      # Import here to avoid circular imports
      from jax.experimental.key_reuse._core import call_impl_with_key_reuse_checks  #
 pytype: disable=import-error
      return call_impl_with_key_reuse_checks(primitive, primitive.impl, f, *tracers,
**params)
    else:
      return primitive.impl(f, *tracers, **params)
```

✓ torch.jit.trace

torch.jit.trace通过TraceState记录算子的执行，最后生成Graph/Block/Node节点，最后优化计算图。

# AI框架的JIT方式

- 解析型

通过Python的decorate能力，捕获函数调用，通过ast.parse将Python函数的字节码解析成抽象语法树，通过对图的节点参数输入输出类型的解析推导（infer）实现静态图的编译优化。对字节码不会进行修改。

优点：

✓ 可以感知到Python语法，流程分析中不再局限于算子的分析，对于分支、循环都可以解析处理。

✓ 捕获成功的话，可以形成完整的整图。

缺点：

✓ Python过于灵活，支持这部分语法不容易。比如：try...except...else...finally, with等。

✓ 构图不一定能够成功。

✓ 由于语法的支持度不全，易用性下降。

- 案例

✓ Triton

Triton定义了15类Ops，在程序运行过程中，通过ast.parse生成python的ast，每个算子通过pybind映射到C++侧，TritonOpBuilder通过MLIR，生成对应方言下的算子节点，最后通过优化pass生成静态图。

✓ torch.jit.script

torch.jit.script通过ast.parse，解析并生成Graph/Block/Node节点，最后优化计算图，后端采用一个解释器解释运行。

# AI框架的JIT方式

- 解释执行型

模拟Python虚拟机的执行，通过对Python的字节码的解释执行（不生成AST，通过字节码），对算子节点采用推导或执行的方式，得到输出FakeTensor，生成Graph和节点，并交由后端优化，同时调整修改字节码（适配副作用等），最后形成动态图（python解释执行）和静态图（后端运行）的混合执行。

优点：

✓ 可以感知到Python语法，流程分析中不再局限于算子的分析，对于分支、循环都可以解析处理。

✓ 通过修改字节码，Python语法支持灵活。

✓ 不会存在图捕获失败的错误报告。

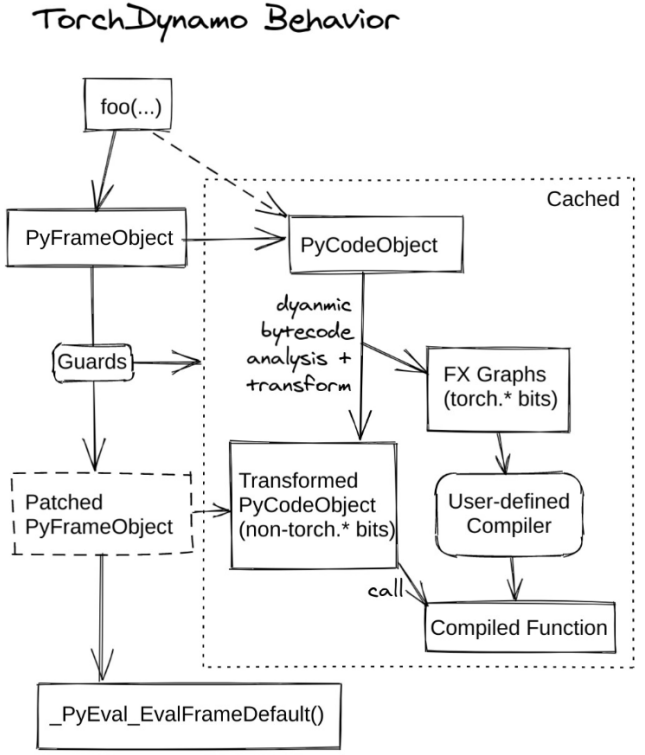
缺点：

✓ 多子图且动静态图混合执行。

- 案例

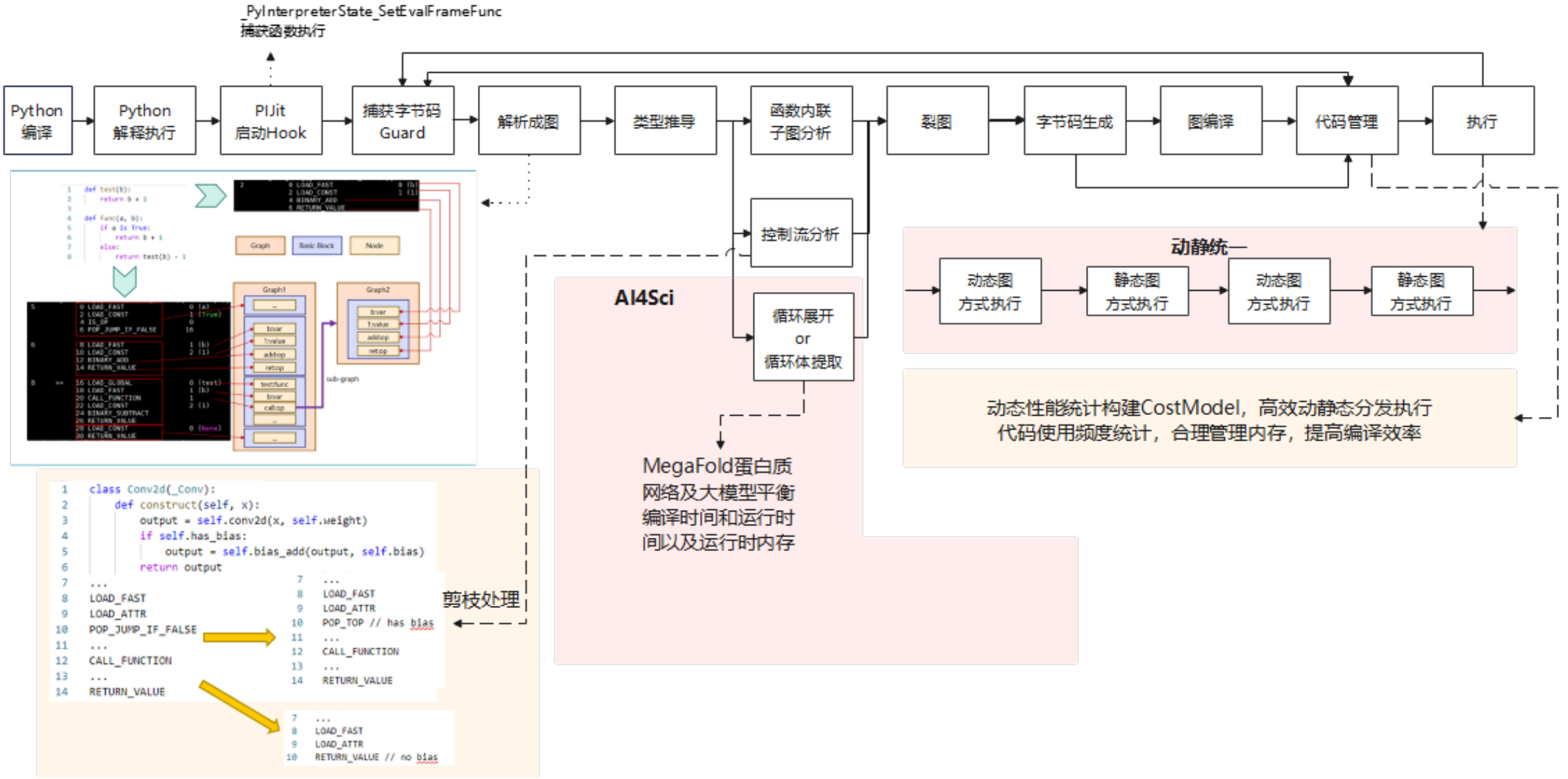
✓ torch.compile

torch.compile通过decorate捕获需要优化的函数，并通过字节码模拟Python的解释执行，对于循环展开采用trace的方式，生成torch.fx.Graph，交由后续pass继续优化。



# JIT技术解决AI大模型动态图性能所面临的挑战和方案

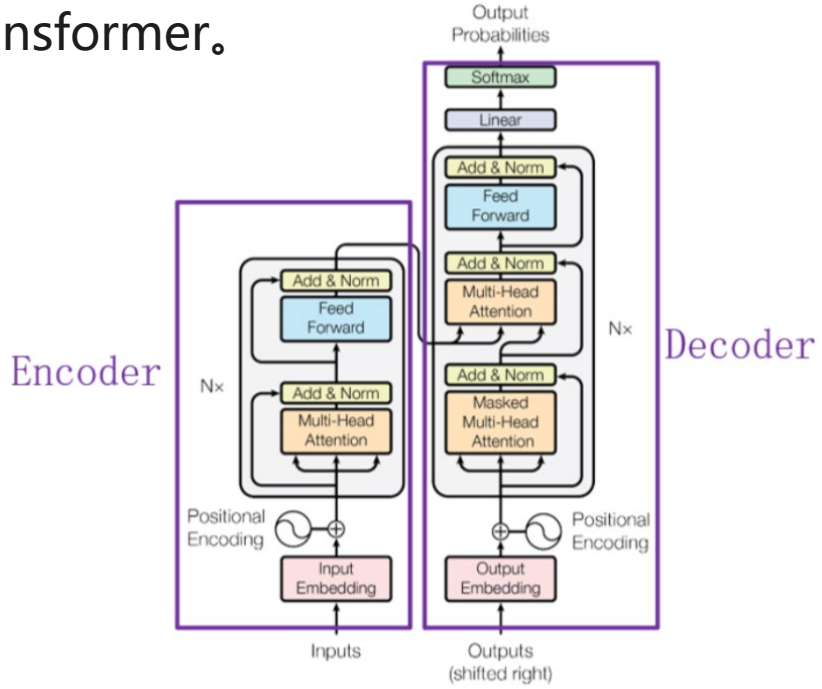
- PIJIT (Python JIT)：一个解决AI大模型动态图性能的JIT方案



# JIT技术解决AI大模型动态图性能所面临的挑战和方案

AI大模型的挑战：

- 模型参数量巨大，但是结构单一，主体是Transformer。




问题：

- 子图复用 v.s. 整图运行 => 循环展开的策略 => trace整个循环 v.s. 循环体单独成图

- 语言模型的输入长度往往是动态的，导致中间结构动态性，需要支持动态shape。

```
class Attention(nn.Module):
def forward(…):
        …
        xq, xk = apply_rotary_emb(xq, xk, freqs_cis=freqs_cis)
        self.cache_k = self.cache_k.to(xq)
        self.cache_v = self.cache_v.to(xq)
        self.cache_k[:bsz, start_pos : start_pos + seqlen] = xk
        self.cache_v[:bsz, start_pos : start_pos + seqlen] = xv
        keys = self.cache_k[:bsz, : start_pos + seqlen]
        values = self.cache_v[:bsz, : start_pos + seqlen]
        …
```

有两种方式：

1.用户设置手工给输入输出设置动态shape或者symbolic shape。

2.通过运行时，动态监测到shape的变化和规律。对不同shape进行合并和优化。

比如：

tensor([1]), tensor([1,1]), tensor([1,1,1])=>tensor([-1])
tensor([[1]]), tensor([[1,1], [1,1]]) => tensor(Sym_a x Sym_a) or duck size

# JIT技术解决AI大模型动态图性能所面临的挑战和方案

- 大模型存在多层循环嵌套情况，当循环体中存在分支控制流，如果采用剪枝处理，则会带来Guard判断深度过深的问题。

```
class Attention(nn.Module):
def forward(…):
        …
        xq = xq.transpose(1, 2)  # (bs, n_local_heads, seqlen, head_dim)
        keys = keys.transpose(1, 2) # (bs, n_local_heads, cache_len + seqlen, head_dim)
        values = values.transpose(1, 2) # (bs, n_local_heads, cache_len + seqlen, head_dim)
        scores = torch.matmul(xq, keys.transpose(2, 3)) / math.sqrt(self.head_dim)
        if mask is not None:
            scores = scores + mask  # (bs, n_local_heads, seqlen, cache_len + seqlen)
        scores = F.softmax(scores.float(), dim=-1).type_as(xq)
        output = torch.matmul(scores, values)  # (bs, n_local_heads, seqlen, head_dim)
        output = output.transpose(1, 2).contiguous().view(bsz, seqlen, -1)
        return self.wo(output)
```

方案：

✓ 将分支处理交由后端处理。

✓ 可以通过常量折叠，常量传播优化分支。

✓ 对Guard条件进行优化。

- 多图的有效管理调度

✓ 许多相似子图需要去重

✓ 当资源不够时，子图管理器可以通过GC回收使用频度小的子图。

✓ 子图运行时，如果从运行性能角度讲，并不一定对于EagerMode的执行性能一定比静态图性能差，需要CostModel进行评估，从而合理调度。

- 其他在传统AI模型上使用JIT也需要解决的问题：

✓ 副作用处理：在静态图闭包外，添加字节码对副作用的处理。

✓ Python语法的静态化：对于无法入图的python语法采用修改字节码的方式，最大化静态图。

✓ 自动化超参调优

# Thank you.

欢迎线下沟通交流

吴江铭/648445
wujiangming2@huawei.com