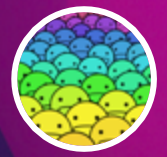
AI编译器-系列之PyTorch

TorchDynamo



ZOMI

Talk Overview

1. **PyTroch 2.0 新特性**
   - 2.0 新特性回顾
   - PyTorch 2.0 安装与新特性使用
   - PyTorch 2.0 对厂商的启发和思考
2. **TorchDynamo 解读**
   - TorchDynamo 特性
   - TorchDynamo 实现方案
3. **AOTAutograd 解读**
   - AOTAutograd 效果
   - AOTAutograd 实现方案
4. **TorchInductor 新特性**
   - Triton 使用解读
   - Triton 深度剖析

Talk Overview

1. TorchDynamo 解读

- PyTorch 获取计算图的方式
- PyTorch JIT Script 原理
- PyTorch FX 原理
- PyTorch Lazy Tensor 原理
- TorchDynamo 原理

图模式的尝试回顾

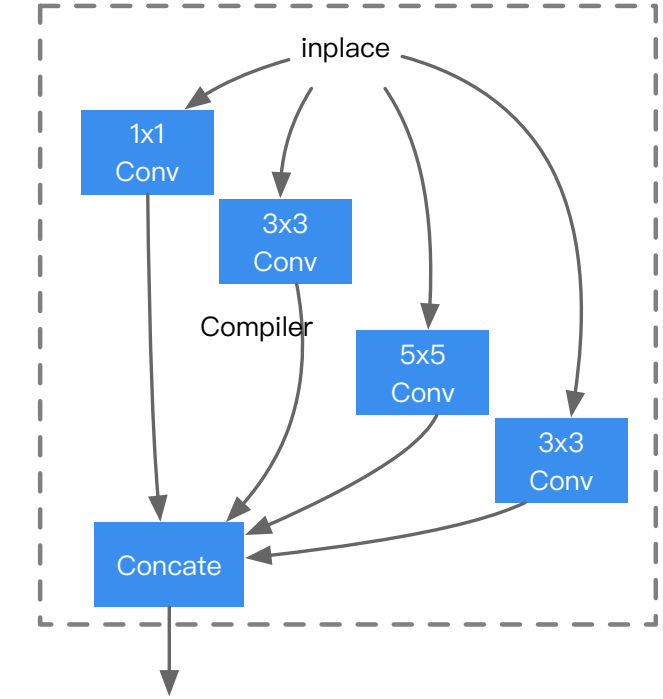
Purpose

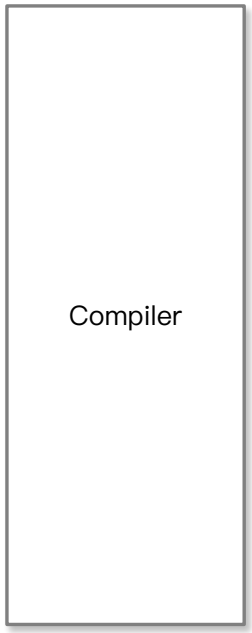
```python
import torch
import torch_xla.core.xla_model as xm


dev = xm.xla_device()

x1 = torch.conv2d((input, 3))
x2 = torch.conv2d((3, 8))
x3 = torch.conv2d((3, 16))
x4 = torch.conv2d((3, 24))
out = torch.concat(x1 , x2, x3, x4)

```





Python Code

Computational Graph

Compiler

Comparation

特性	内容	适用场景	技术方式
TorchScript	TorchScript 的主要用途是进行模型部署，需要记录生成一个便于推理优化的 IR，对计算图的编辑通常都是面向性能提升等等，不会给模型本身添加新的功能。	推理和部署静态图优化	AST + Trace
Torch FX	FX 的主要用途是进行python->python的翻译，它的 IR 中节点类型更简单，比如函数调用、属性提取等等，这样的 IR 学习成本更低更容易编辑。使用 FX 来编辑图通常是为了实现某种特定功能，比如给模型插入量化节点等，避免手动编辑网络造成的重复劳动。	修改正向图	Trace
LazyTensor	采用 tensor 不被用户观察那么就可以缓存计算指令的方式，隐式构图+优化，用户不用标记构图的代码范围，对用户的代码限制小，是一种能优化就优化、不能优化也能执行的思路，偏向于易用性	针对NPU子图编译优化	Trace
TorchDynamo	TorchDynamo 是 PyTorch 新实验的 JIT 编译接口，支持使用 Python 在运行时修改动态执行逻辑，修改的时机是 CPython 的 ByteCode 执行前。思想类似 DynamoRIO 项目，DynamoRIO 可以动态的修改 x86 机器码。	动静统一 + 编译优化	Python 解析修改

TorchDynamo 原理

What makes TorchDynamo sound and out-of-the-box?

Partial graph capture

Ability to skip unwanted parts of eager

Guarded graphs

Ability to check if captured graph is valid for execution

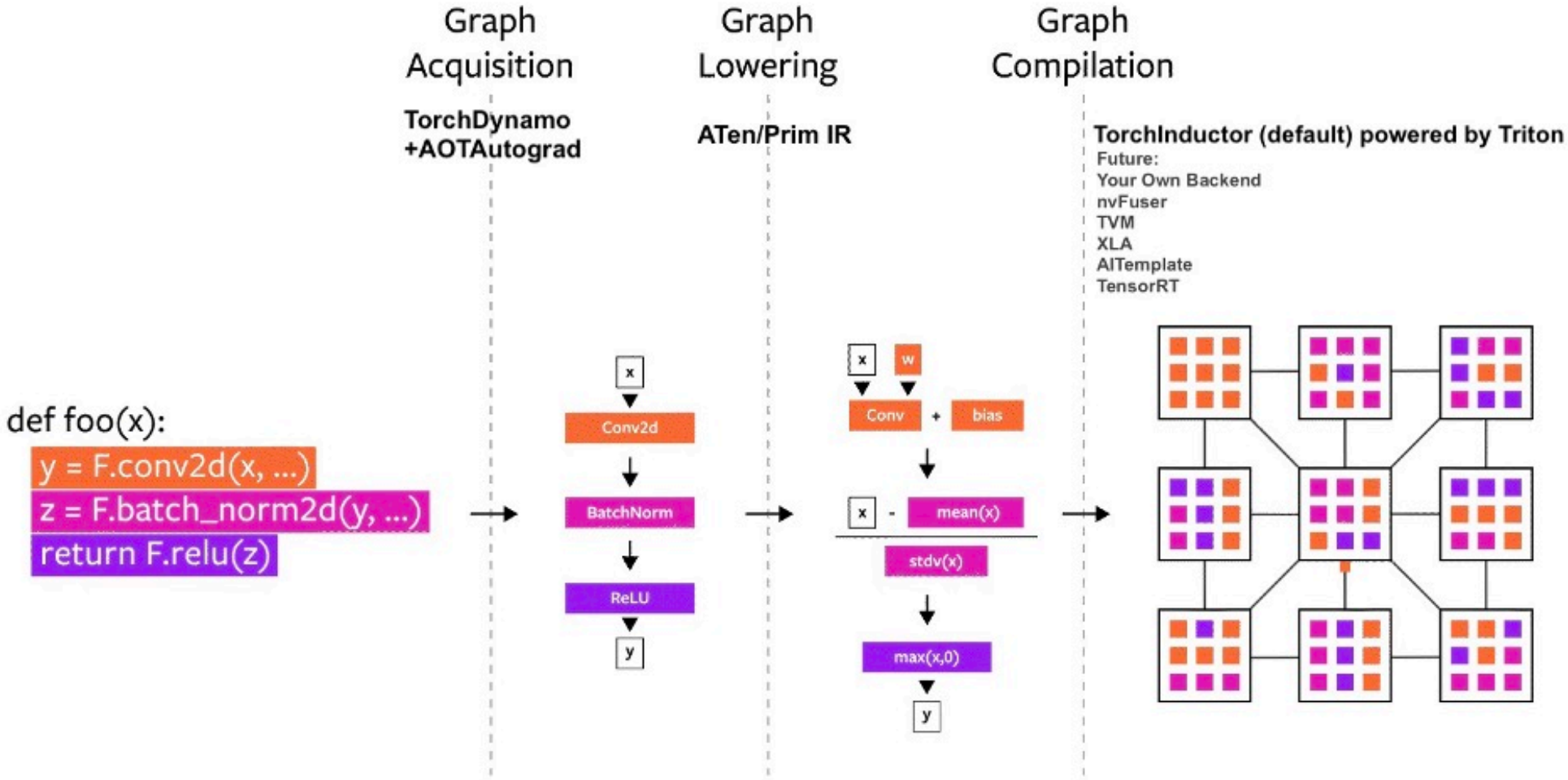
Just-in-time recapture

Recapture a graph if captured graph is invalid for execution

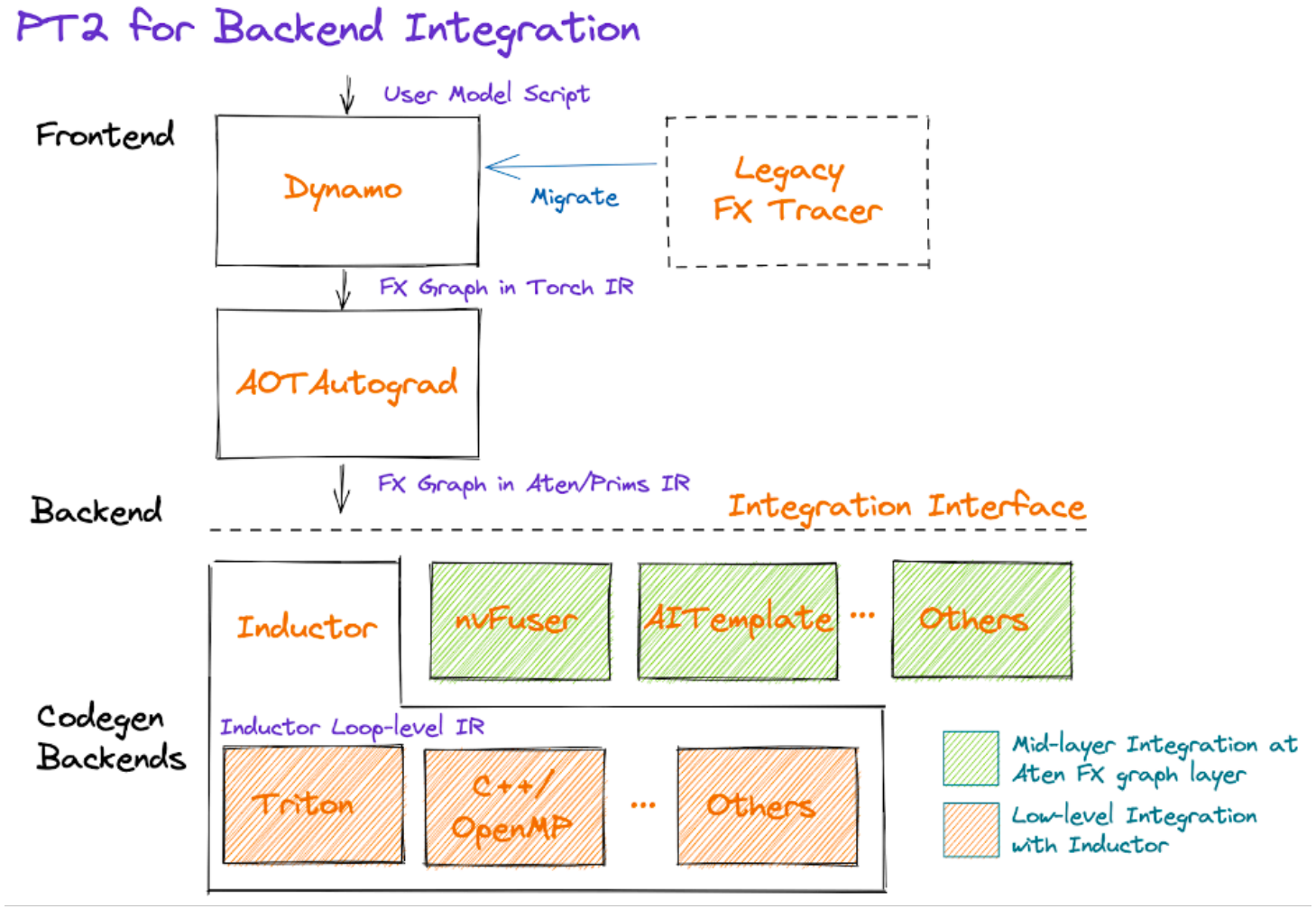
TorchDynamo Today

7k+	20+	1+	30+%
Crawled GitHub Models	Inference Backends	Training backends	Geomean OOTB speedup (Models - TB, TIMM, HF) FP32, AMP A100 GPU

PyTorch 编译模式

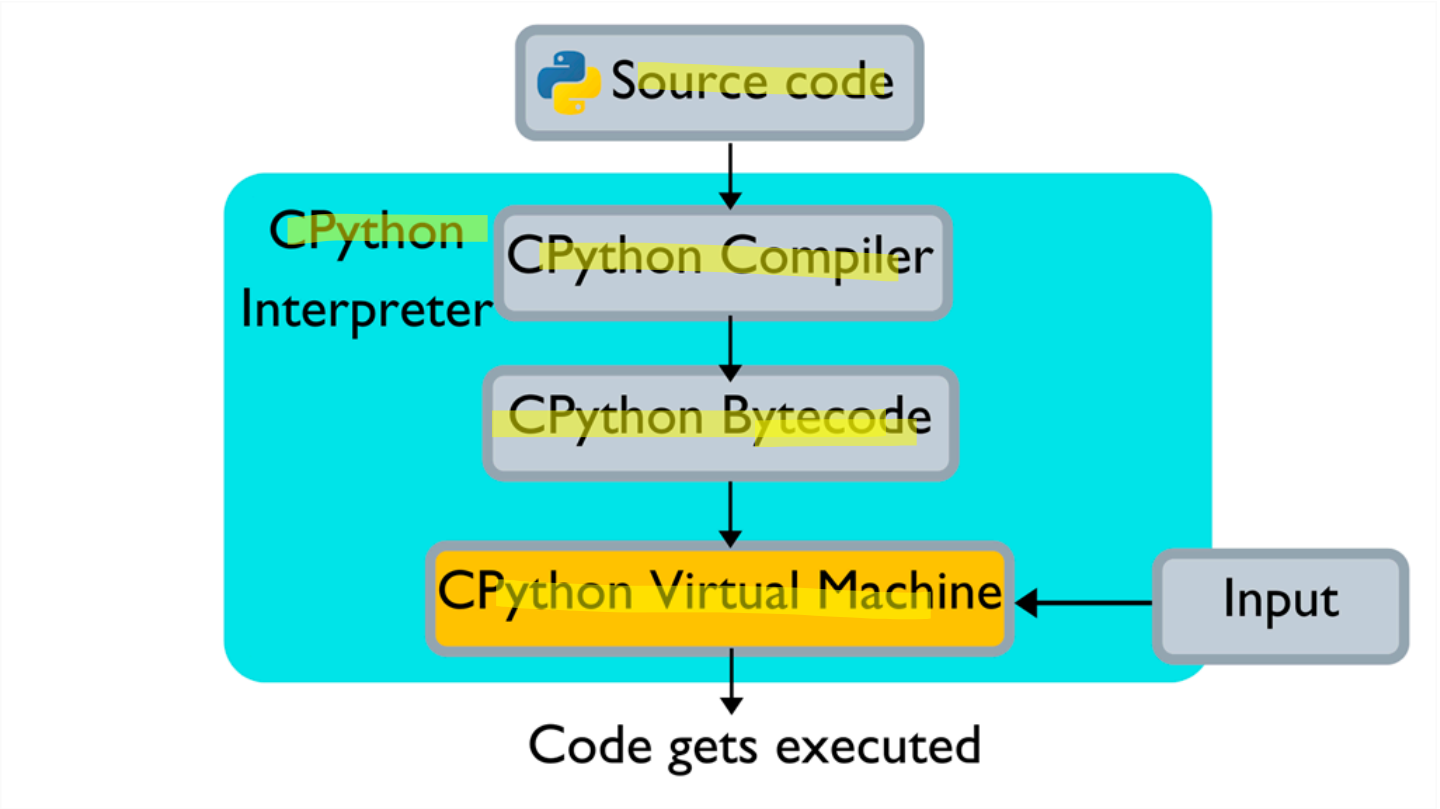


PyTorch 编译模式



Python 执行机制

1. **CPython：** CPython 混合了编译和解释功能，将Python源代码首先被编译成一系列中间字节码，然后由CPython虚拟机内部 while 循环不断匹配字节码并执行对应字节码指令 case 分支内部的多条C函数。

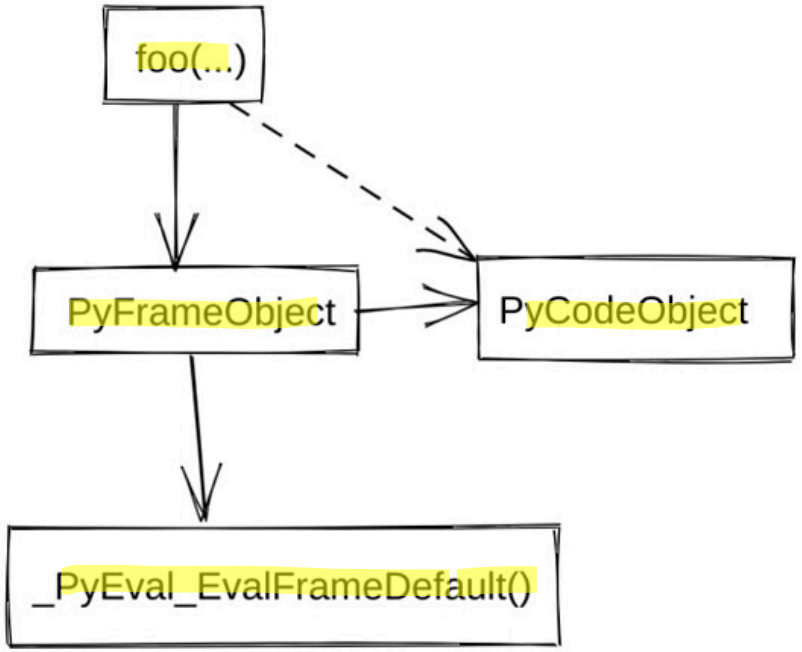


Python 执行机制： PyFrameObject 和 PyCodeObject

1. 每一个PyFrameObject对象都维护着一个PyCodeObject对象，每一个PyFrameObject对象和Python源码中的一段code都是对应的。
2. PyCodeObject对象包含了程序中的静态信息，然而有一点PyCodeObject对象没有包含，那就是关于程序运行时的动态信息——执行环境。
3. 在Python真正执行的时候，它的虚拟机实际上面对的并不是一个PyCodeObject对象，而是另外一个对象——PyFrameObject。
4. 在Python实际执行的过程中，会产生很多PyFrameObject对象，而这些对象会被链接起来，形成一条执行环境链表。
5. PyFrameObject对象是对x86机器上单个栈帧活动的模拟，既然在x86的单个栈帧中，包含了计算所需的内存空间、寄存器地址指针、运行栈顺序、执行环境等信息。

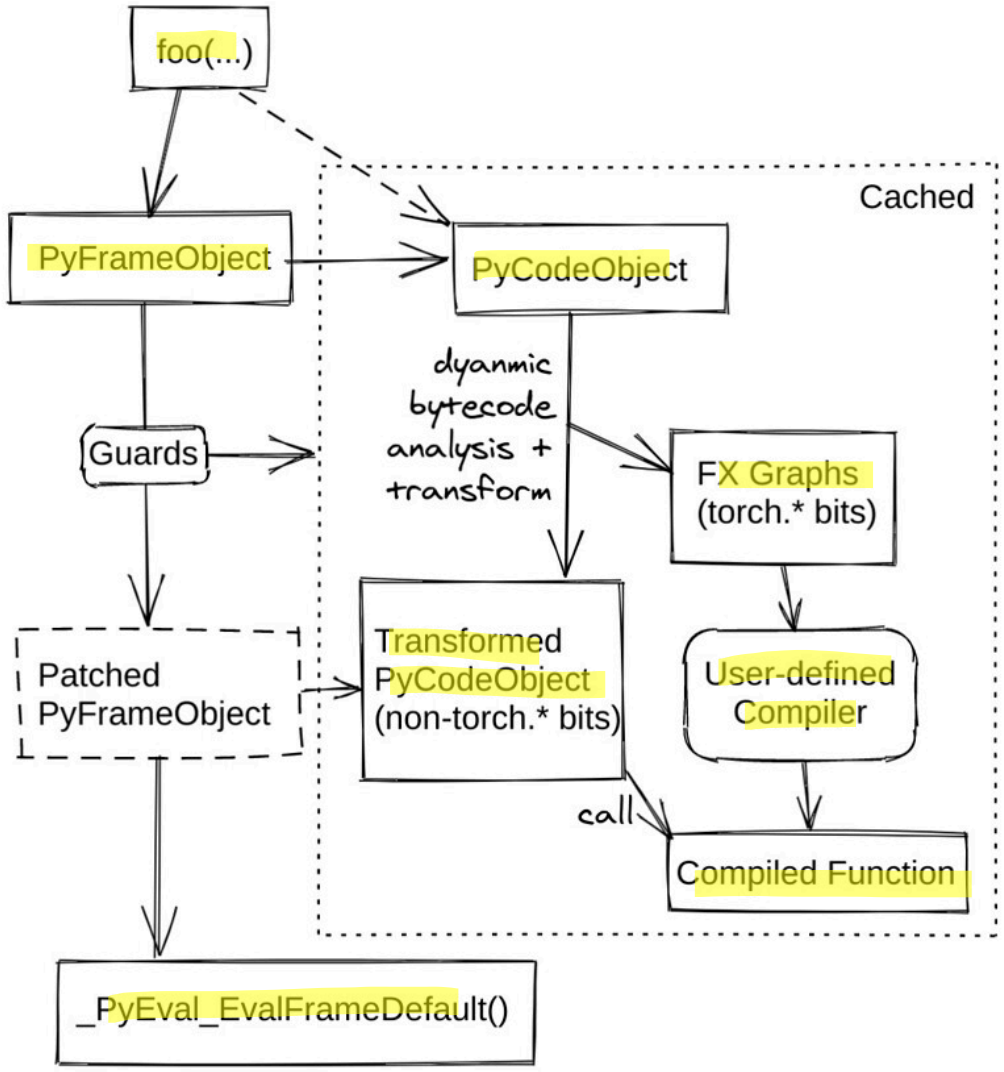
Python 执行机制

- 在 CPython 解释器 Interpreter 中运行Python程序。虽然Python代码被转换为字节码byteCode，但是这个字节码并不在CPU上运行，而是由程序本身执行。这个解释器程序的核心是 _PyEval_EvalFrameDefault。



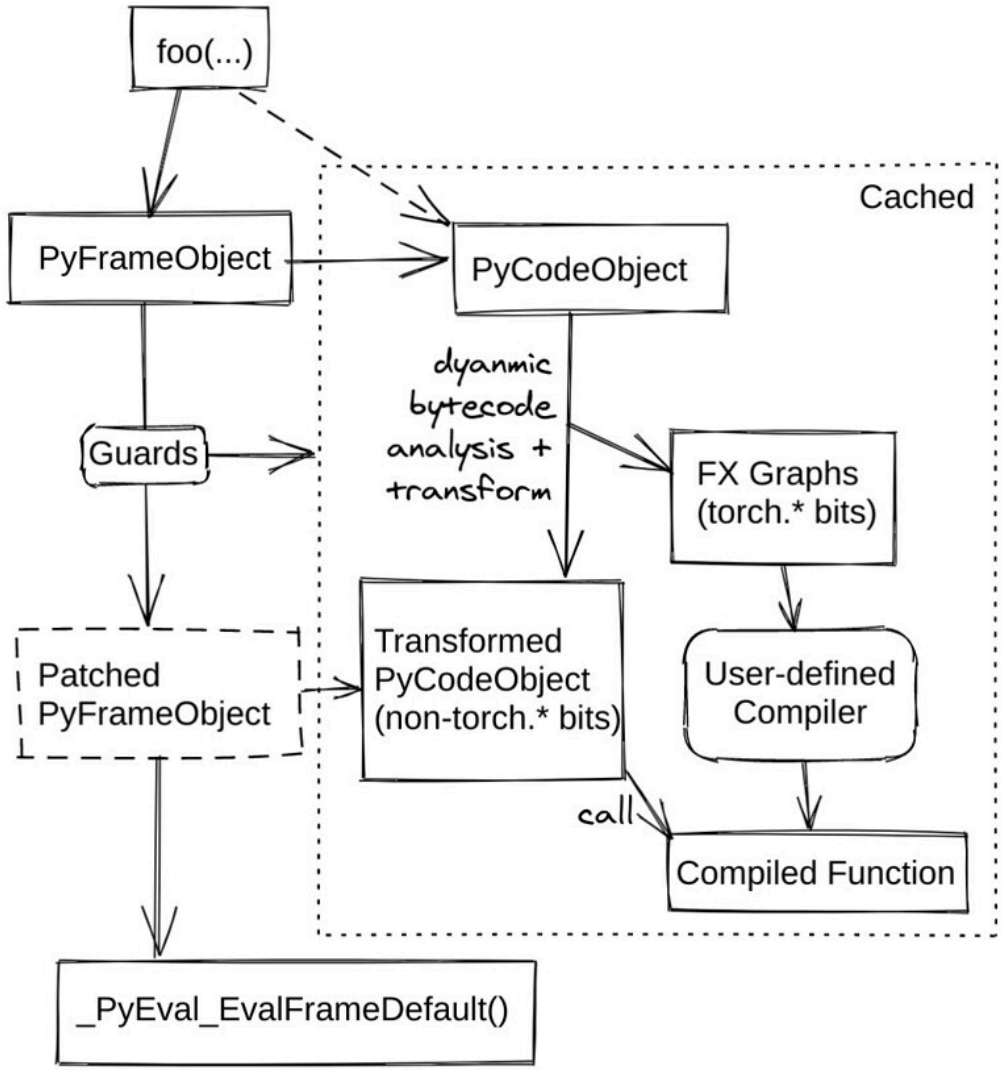
Dynamo 实现原理

- 在 Python 解释器自动把动态 ByteCode 进行捕抓，转换成为为 TorchFX IR。



Dynamo 实现原理

- TorchDynamo is an early experiment that radically rethinks the approach for recapturing these optimization opportunities. It hooks into the frame evaluation API in CPython to dynamically modify Python bytecode right before it is executed.
- This is analogous to what DynamoRIO does by dynamically modifying x86 machine code.
- TorchDynamo dynamically rewrites Python bytecode in order to extract sequences of PyTorch operations into an FX Graph which is then just-in-time compiled with a user-defined compiler.
- It creates FX Graph through bytecode analysis, not tracing, and is designed to generating smaller graph fragments that can be mixed with Python execution.



TORCH.FX NODE SEMANTICS

- torch.fx represents programs in a DAG-based IR, which contains a linear series of Node objects representing operations. Nodes have a string opcode, describing what type of operation the Node represents. Nodes have an associated target, which is the call target for call nodes (call_module, call_function, and call_method).

A.1 Opcode Meanings

Opcode	Meaning
`placeholder`	Function Input
`call_method`	Call method on `args[0]`
`call_module`	Call module specified by `target`
`call_function`	Call function specified by `target`
`get_attr`	Retrieve attribute specified by `target`
`output`	Return statement; return `args[0]`

A.2 `args/kwargs` Behavior

Opcode	`args/kwargs` Behavior
`placeholder`	Empty or `args[0]` = default value
`call_method`	Python calling convention; `args[0]` is self
`call_module`	Python calling convention; `target` is self
`call_function`	Python calling convention; `target` is self
`get_attr`	Empty
`output`	`args[0]` is the return value

TORCH.FX IR

1. placeholder represents a function input. The name attribute specifies the name this value will take on. target is similarly the name of the argument. args holds either: 1) nothing, or 2) a single argument denoting the default parameter of the function input. kwargs is don't-care. Placeholders correspond to the function parameters (e.g. x) in the graph printout.
2. get_attr retrieves a parameter from the module hierarchy. name is similarly the name the result of the fetch is assigned to. target is the fully-qualified name of the parameter's position in the module hierarchy. args and kwargs are don't-care
3. call_function applies a free function to some values. name is similarly the name of the value to assign to. target is the function to be applied. args and kwargs represent the arguments to the function, following the Python calling convention
4. call_module applies a module in the module hierarchy's forward() method to given arguments. name is as previous. target is the fully-qualified name of the module in the module hierarchy to call. args and kwargs represent the arguments to invoke the module on, including the self argument.
5. call_method calls a method on a value. name is as similar. target is the string name of the method to apply to the self argument. args and kwargs represent the arguments to invoke the module on, including the self argument
6. output contains the output of the traced function in its args[0] attribute. This corresponds to the "return" statement in the Graph printout.

Results in the following original Python bytecode

```
0  LOAD_FAST 0 (a)
2  LOAD_FAST 1 (b)
4  BINARY_ADD
6  STORE_FAST 2 (x)

8  LOAD_FAST 2 (x)
10 LOAD_CONST 1 (2.0)
12 BINARY_TRUE_DIVIDE
14 STORE_FAST 2 (x)

16 LOAD_FAST 2 (x)
18 LOAD_METHOD 0 (sum)
20 CALL_METHOD 0
22 LOAD_CONST 2 (0)
24 COMPARE_OP 0 (<)
26 POP_JUMP_IF_FALSE 36

28 LOAD_FAST 2 (x)
30 LOAD_CONST 3 (-1.0)
32 BINARY_MULTIPLY
34 RETURN_VALUE

36 LOAD_FAST 2 (x)
38 RETURN_VALUE
```

TorchDynamo dynamically rewrites that bytecode as follows

```
0  LOAD_GLOBAL 1 (__compiled_fn_0)
2  LOAD_FAST 0 (a)
4  LOAD_FAST 1 (b)
6  CALL_FUNCTION 2
8  UNPACK_SEQUENCE 2
10 STORE_FAST 2 (x)
12 POP_JUMP_IF_FALSE 22

14 LOAD_GLOBAL 2 (__compiled_fn_1)
16 LOAD_FAST 2 (x)
18 CALL_FUNCTION 1
20 RETURN_VALUE

22 LOAD_FAST 2 (x)
24 RETURN_VALUE
```

torch.compile on our original goal on 163 models

93%

Works out of the box

43%

Faster in Aggregate compared to Eager mode on an NVIDIA A100

Dynamo Pros and Cons

This approach has many advantages:

- It supports all Python because it can easily fall back to running the original bytecode. It depends on a fast eager mode in order to work properly, because the goal is to enhance eager mode rather than replace it.
- It is extremely low overhead, where it is possible to remove Python overheads from the original program by intercepting things at the very top of the stack.
- It does not introduce any added latency by deferring execution.

Inference

1. TorchDynamo: An Experiment in Dynamic Python Bytecode Transformation. https://ev-discuss.pytorch.org/t/torchdynamo-an-experiment-in-dynamic-python-bytecode-transformation/361
2. The nuances of PyTorch Graph Capture. https://dev-discuss.pytorch.org/t/the-nuances-of-pytorch-graph-capture/501
3. TORCHDYNAMO AND TORCHINDUCTOR TUTORIAL. https://pytorch.org/tutorials/intermediate/dynamo_tutorial.html