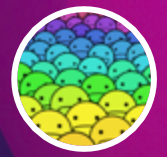
AI编译器-系列之PyTorch

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

PyTorch
获取计算图

PyTorch Eager Mode

A graph is created on the fly

```
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))
```

WE SPENT 5 YEARS ON R&D ANSWERING:

How to capture dynamic graphs without compromising user-experience?

torch.compile(model)

Why should you care about PyTorch graph capture?

CHIP DESIGNERS

Eager-mode execution is considered prohibitively costly for accelerators

PRODUCTION ENGINEERS

Most PyTorch inference deployments are exported out of Eager via graph capture

COMPILER ENGINEERS

No graph, no compiler

获取计算图的主要方式

- **基于追踪Trace：**直接执行用户代码，记录下算子调用序列，将算子调用序列保存为静态图，执行中脱离前端语言环境，由运行时按照静态图逻辑执行；
- **基于源代码解析：**以宿主语言的抽象语法树（AST）为输入，转化为内部语法树，经过别名分析，SSA（static single value assignment），类型推断等Pass，转换为计算图表示；

获取计算图的主要方式 (I)

PT FX

- **基于追踪Trace**：直接执行用户代码，记录下算子调用序列，将算子调用序列保存为静态图，执行中脱离前端语言环境，由运行时按照静态图逻辑执行；

```
import torch
import torch.fx

class MyModule(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.param = torch.nn.Parameter(torch.rand(3, 4))
        self.linear = torch.nn.Linear(4, 5)

    def forward(self, x):
        return torch.topk(torch.sum(
            self.linear(x + self.linear.weight).relu(), dim=-1), 3)

m = MyModule()
gm = torch.fx.symbolic_trace(m)

gm.graph.print_tabular()
```

获取计算图的主要方式 (I)

PT FX

- **基于追踪Trace**：直接执行用户代码，记录下算子调用序列，将算子调用序列保存为静态图，执行中脱离前端语言环境，由运行时按照静态图逻辑执行；

优点

- 能够更广泛地支持宿主语言中的各种动态控制流语句；

缺点

- 执行场景受限，只能保留程序有限执行轨迹并线性化，静态图失去源程序完整控制结构；

获取计算图的主要方式 (II)

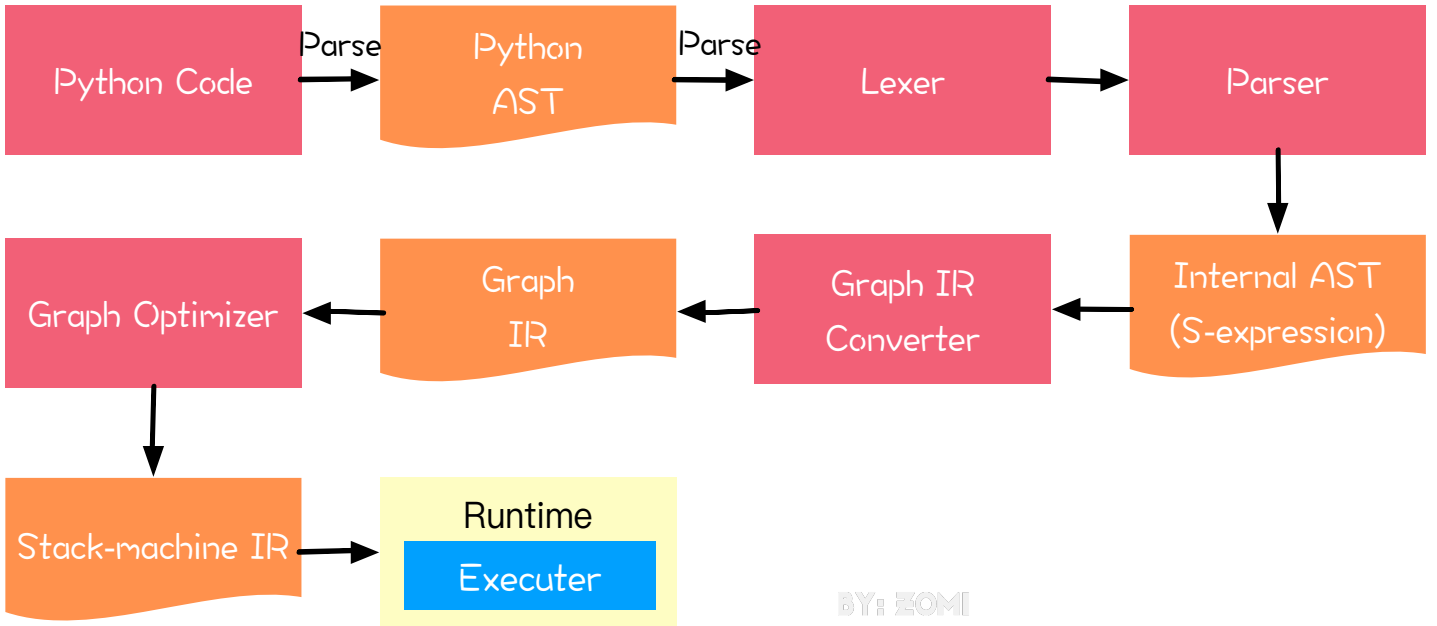
PT JIT

- **基于源代码解析：**以宿主语言的抽象语法树（AST）为输入，转化为内部语法树，经过别名分析，SSA（static single value assignment），类型推断等Pass，转换为计算图表示；

```python
import torch
from torch import Tensor

@torch.jit.script
def foo1(x: Tensor, y: Tensor, z: Tensor):
    if x < y:
        s = x + y
    else:
        s = torch.square(y)
    return s

@torch.jit.script
def foo2(s: Tensor):
    for i in torch.range(10):
        s += i
    return s
```



获取计算图的主要方式 (II)

- **基于源代码解析：**以宿主语言的抽象语法树（AST）为输入，转化为内部语法树，经过别名分析，SSA（static single value assignment），类型推断等Pass，转换为计算图表示；

优点

- 能够更广泛地支持宿主语言中的各种动态控制流语句

缺点

- 后端实现和硬件实现会对静态图表示进行限制和约束，多硬件需要切分多后端执行逻辑；
- 宿主语言的控制流语句并不总是能成功映射到后端运行时系统的静态图表示；
- 遇到过度灵活的动态控制流语句，运行时会退回到由前端语言跨语言调用驱动后端执行；

PyTorch 编译的尝试

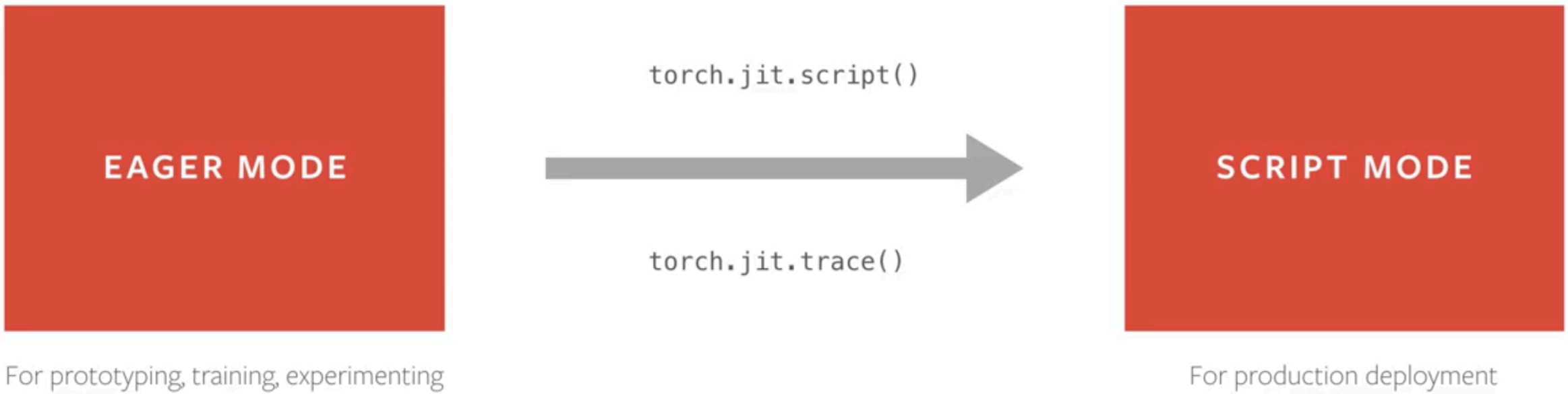
- Torch FX
- TorchScript
- Lazy Tensor
- TorchDynamo

TorchScript 原理

TorchScript

- TorchScript 是 Python 的静态类型子集，可以通过 @torch.jit.script 装饰器进行Python源码解析或者通过基于跟踪（ tracing base ）从python代码自动生成。
- TorchScript 从 PyTorch 代码创建可序列化和可优化模型的方法。任何 TorchScript 程序都可以从 Python 进程中保存，并加载到没有 Python 依赖的进程中。

TOOLS TO TRANSITION FROM EAGER TO SCRIPT



For prototyping, training, experimenting

For production deployment

EAGER TO SCRIPT MODE WITH `torch.jit.trace()`

Take an existing eager model, and provide example inputs.

The tracer runs the function, recording the tensor operations performed.

We turn the recording into a TorchScript module.

- Can reuse existing eager model code
- ⚠ Control-flow and data structures are ignored

```
import torch
import torchvision

def foo(x, y):
  return 2 * x + y

# trace a model by providing example inputs
traced_foo = torch.jit.trace(foo,
                             (torch.rand(3), torch.rand(3)))

traced_resnet = torch.jit.trace(torchvision.models.resnet18(),
                                torch.rand(1, 3, 224, 224))
```

EAGER TO SCRIPT MODE WITH `torch.jit.script()`

Write model directly in TorchScript, a high-performance subset of Python.

Pass instance of your model to `torch.jit.script()`

- Control-flow is preserved
- `print` statements can be used for debugging
- Remove the script() call to debug as a standard PyTorch module

```
class RNN(torch.nn.Module):
  def __init__(self, W_h, U_h, W_y, b_h, b_y):
    super(RNN, self).__init__()
    self.W_h = nn.Parameter(W_h)
    self.U_h = nn.Parameter(U_h)
    self.W_y = nn.Parameter(W_y)
    self.b_h = nn.Parameter(b_h)
    self.b_y = nn.Parameter(b_y)

  def forward(self, x, h):
    y = []
    for t in range(x.size(0)):
      h = torch.tanh(x[t] @ self.W_h + h @ self.U_h + self.b_h)
      y += [torch.tanh(h @ self.W_y + self.b_y)]
      if t % 10 == 0:
        print("stats: ", h.mean(), h.var())
    return torch.stack(y), h

rnn = RNN(torch.randn(3, 4), torch.randn(4, 4), torch.randn(4, 4),
torch.randn(4), torch.randn(4))
scripted = torch.jit.script(rnn)
```

EXPORTING A MODEL TO PRODUCTION

TorchScript models can be saved a model archive and loaded to run in PyTorch's just-in-time (JIT) compiler instead of the CPython interpreter.

C++ Tensor APIs support bindings to a wide range of languages and deployment environments.

The same TorchScript models can also be loaded in the PyTorch Mobile runtime.

```python
# Python: save model
traced_resnet = torch.jit.trace(torchvision.models.resnet18(),
                                torch.rand(1, 3, 224, 224))
traced_resnet.save("serialized_resnet.zip")
```

```cpp
// C++: load model
auto module = torch::jit::load("serialized_resnet.zip");
auto example = torch::rand({1, 3, 224, 224});

// Execute `forward()` using the PyTorch JIT
auto output = module->forward({example}).toTensor();
std::cout << output.slice(1, 0, 5) << '\n';
```

PYTORCH JIT INTERMEDIATE REPRESENTATION

```
graph(%0 : Float(3, 10), %1 : Float(3, 20), %2 : Float(3, 20), %3 : Float(80, 10)
      %4 : Float(80, 20), %5 : Float(80), %6 : Float(80)) {
  %7 : Float(10!, 80!) = aten::t(%3)
  %10 : int[] = prim::ListConstruct(3, 80)
  %12 : Float(3!, 80) = aten::expand(%5, %10, 1)
  %15 : Float(3, 80) = aten::addmm(%12, %0, %7, 1, 1)
  %16 : Float(20!, 80!) = aten::t(%4)
  %19 : int[] = prim::ListConstruct(3, 80)
  %21 : Float(3!, 80) = aten::expand(%6, %19, True)
  %24 : Float(3, 80) = aten::addmm(%21, %1, %16, 1, 1)
  %26 : Float(3, 80) = aten::add(%15, %24, 1)
  %29 : Dynamic[] = aten::chunk(%26, 4, 1)
  %30 : Float(3!, 20), %31 : Float(3!, 20), %32 : Float(3!, 20), %33 : Float(3!, 20) = prim::ListUnpack(%29)
  %34 : Float(3, 20) = aten::sigmoid(%30)
  %35 : Float(3, 20) = aten::sigmoid(%31)
  %36 : Float(3, 20) = aten::tanh(%32)
  %37 : Float(3, 20) = aten::sigmoid(%33)
  %38 : Float(3, 20) = aten::mul(%35, %2)
  %39 : Float(3, 20) = aten::mul(%34, %36)
  %41 : Float(3, 20) = aten::add(%38, %39, 1)
  %42 : Float(3, 20) = aten::tanh(%41)
  %43 : Float(3, 20) = aten::mul(%37, %42)
  return (%43, %41);
}
```

PYTORCH JIT INTERMEDIATE REPRESENTATION

```
graph(%0 : Float(3, 10), %1 : Float(3, 20), %2 : Float(3, 20), %3 : Float(80, 10)
      %4 : Float(80, 20), %5 : Float(80), %6 : Float(80)) {
  %7 : Float(10!, 80!) = aten::t(%3)
  %10 : int[] = prim::ListConstruct(3, 80)
  %12 : Float(3!, 80) = aten::expand(%5, %10, 1)
  %15 : Float(3, 80) = ate
  %16 : Float(20!, 80!) =
  %19 : int[] = prim::List
  %21 : Float(3!, 80) = at
  %24 : Float(3, 80) = ate
  %26 : Float(3, 80) = ate
  %29 : Dynamic[] = aten::
  %30 : Float(3!, 20), %31                                          ListUnpack(%29)
  %34 : Float(3, 20) = ate
  %35 : Float(3, 20) = aten::sigmoid(%31)
  %36 : Float(3, 20) = aten::tanh(%32)
  %37 : Float(3, 20) = aten::sigmoid(%33)
  %38 : Float(3, 20) = aten::mul(%35, %2)
  %39 : Float(3, 20) = aten::mul(%34, %36)
  %41 : Float(3, 20) = aten::add(%38, %39, 1)
  %42 : Float(3, 20) = aten::tanh(%41)
  %43 : Float(3, 20) = aten::mul(%37, %42)
  return (%43, %41);
}
```

- Static typing
- Structured control flow (nested ifs and loops)
- Functional by default

WHAT KIND OF OPTIMIZATIONS DO WE WANT TO DO?

Algebraic rewriting
- Constant folding, common subexpression elimination, dead code elimination, etc.

Out-of-order execution
- Re-ordering operations to reduce memory pressure and make efficient use of cache locality

Fusion
- Combining several operators into a single kernel to avoid overheads from round-trips to memory, PCIe, etc

Target-dependent code generation
- Taking parts of the program and compiling them for specific hardware
- Integration ongoing with several code generation frameworks: TVM, Halide, Glow, XLA Algebraic rewriting

Maintaining the same semantics is critical for user experience
- Users should get optimization "for free"!

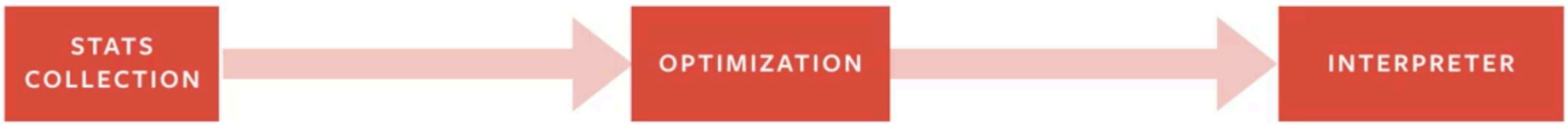
OPTIMIZATION VIA JIT COMPILATION

In production environments, many runtime properties are likely to remain static.

Whenever you have something that can be dynamic, but is likely static, just-in-time compilation may be useful!

OPTIMIZATION VIA JIT COMPILATION

User calls `forward()`



Specialization

- Shape
- Device
- requires_grad

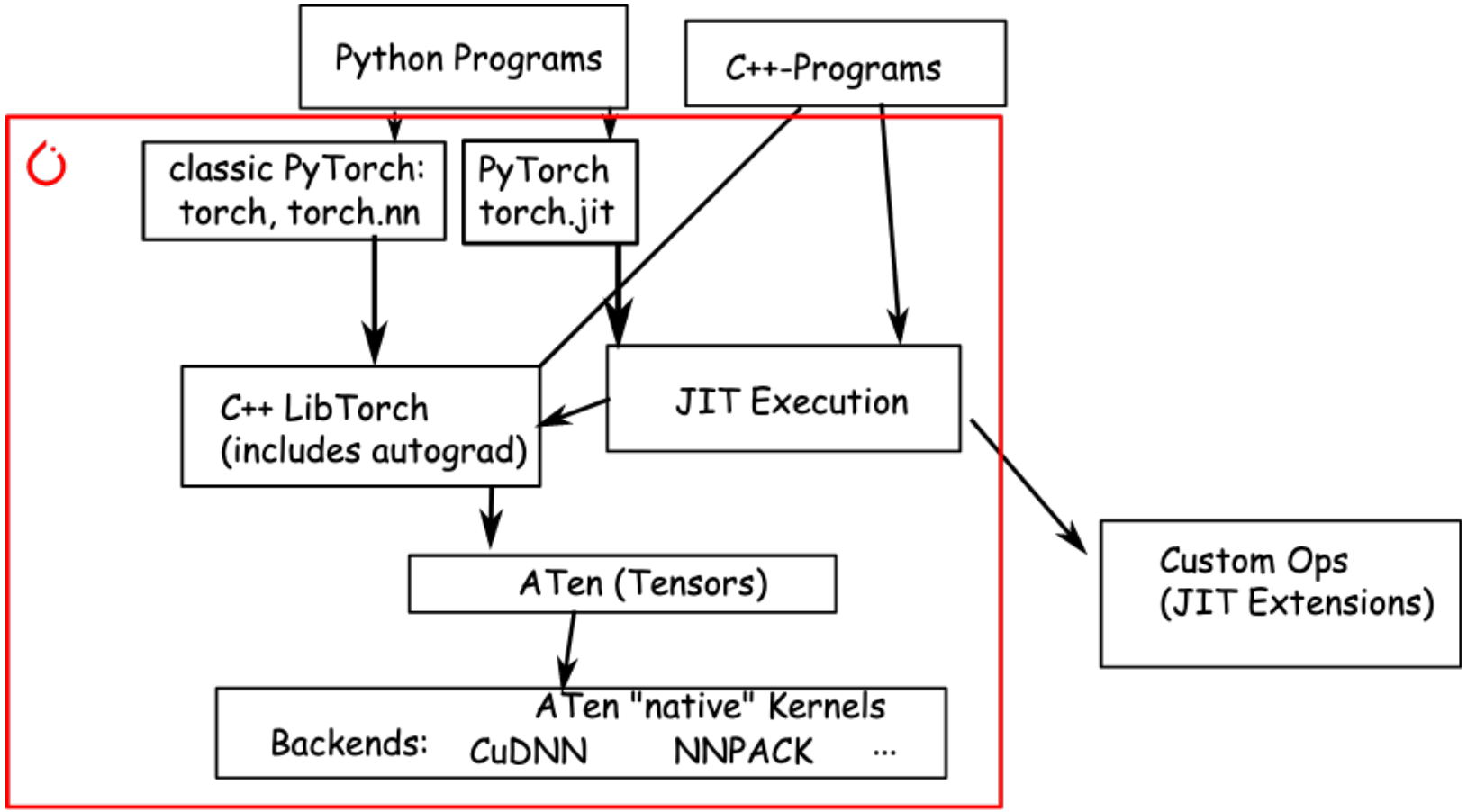
Compiler optimizations

- Fusion (loop, op, etc)
- Algebraic rewriting
- Loop unrolling
- Code generation
- etc...

Execution

- Scheduling
- Parallelism

How JIT work



TorchScript Pros and Cons

Pros

- 能够把动态图转为静态图，对静态图进行编译优化和执行；
- 作为PyTorch关于静态图的第一次尝试；

Cons：

- 静态图只能表达正向图，不能够处理反向图和动态Shape，使用场景有限；
- 静态图IR复杂，难学，对于想修改或者增加优化Pass的开发者来说成本高；