中国科学技术大学
University of Science and Technology of China

面向机器学习的编译

组员：刘硕 毕超 郭雨轩
指导：黄奕桐 张昱

2019.11.20

报告目录

中国科学技术大学
University of Science and Technology of China

简介

- 研究方向：面向深度学习的编译体系
  - 关注：深度学习框架的JIT即时编译器，包括其编译流程、IR表示、优化策略、前向计算和反向微分机制
- 研究框架
  - PyTorch: 开源的Python机器学习库, 底层由C++实现
    2018年12月发布的1.0版本提供即时编译器对模型优化
  - TensorFlow MLIR (Multi-Level Intermediate Representation)：
    是一个统一的IR来统合各个ML框架
  - Zygote.jl：Julia语言的自动微分库
    结合Julia语言的即时编译器实现反向微分计算

中国科学技术大学
University of Science and Technology of China

PyTorch JIT & IR

- PyTorch JIT的编译流程
- PyTorch JIT IR的程序表示
- PyTorch JIT的自动微分
- 总结

PyTorch应用示例

以LeNet-一个简单的数字分类神经网络为例

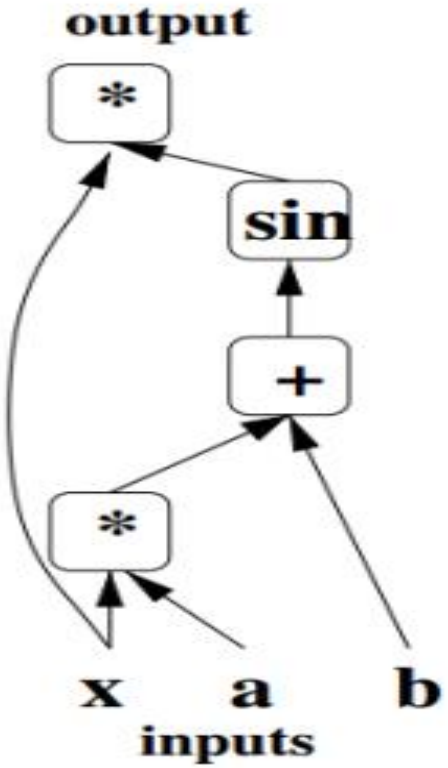
```
def forward(self, x):
    x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
    x = F.max_pool2d(F.relu(self.conv2(x)), 2)
    x = x.view(-1, self.num_flat_features(x))
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    x = self.fc3(x)
return x

output = net(input)
define target
criterion = nn.MSELoss() #pytorch封装的一个损失函数
loss = criterion(output, target)
```

从tensor input 到损失值loss的计算图会被记录下来：

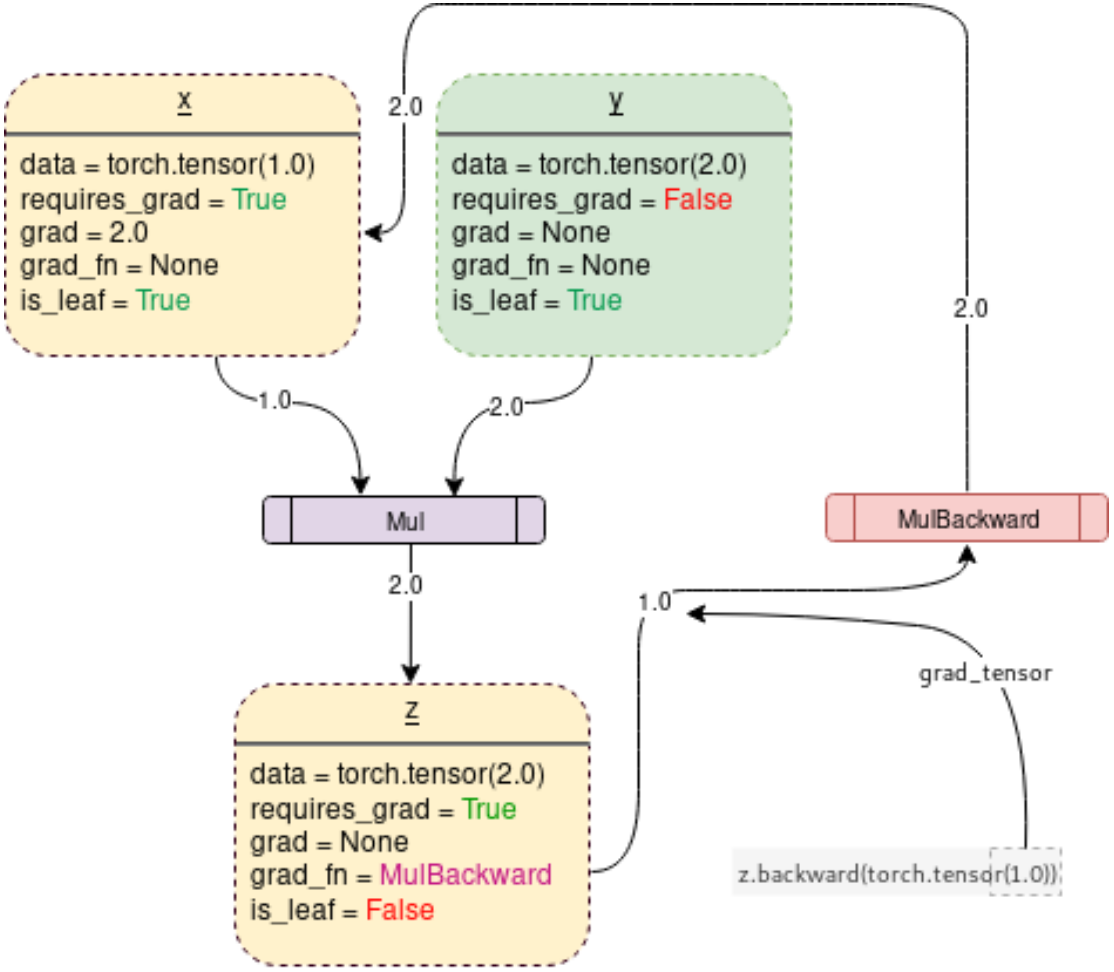
input -> conv2d -> relu -> maxpool2d -> conv2d -> relu -> maxpool2d -> view -> linear -> relu -> linear -> relu -> linear -> MSELoss -> loss

计算图示例

output = x*sin(a*x+b)的计算图

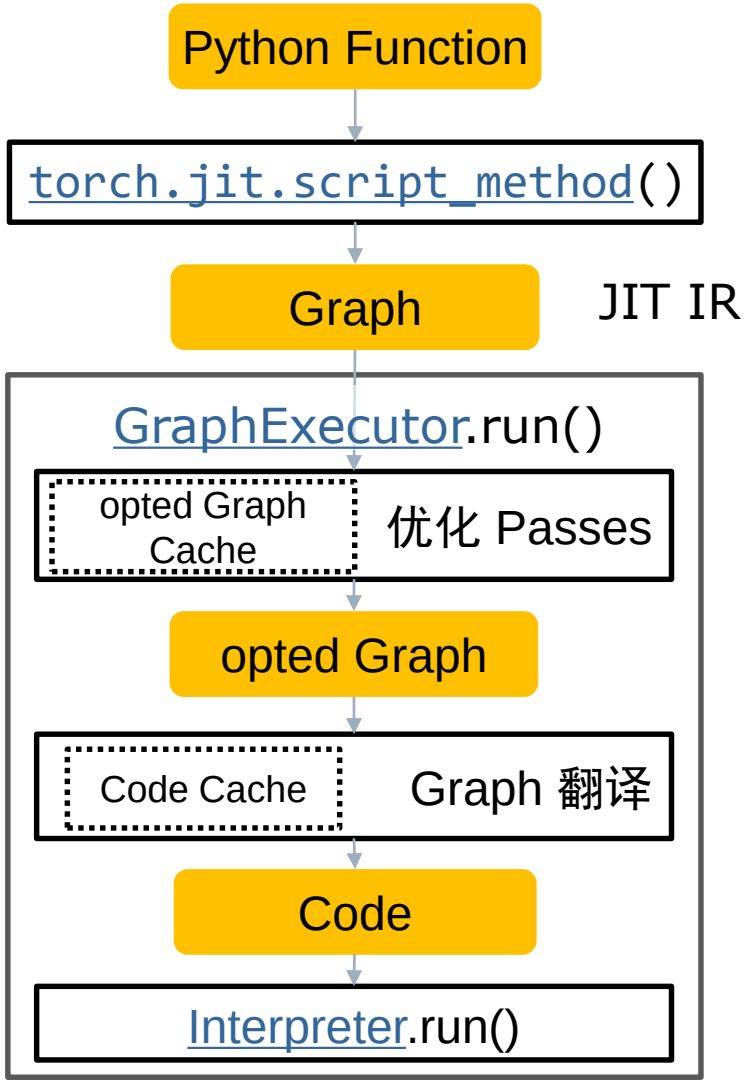


动态创建计算图 z=x*y



引自CSDN：PyTorch的自动求导机制详解

PyTorch JIT执行流程



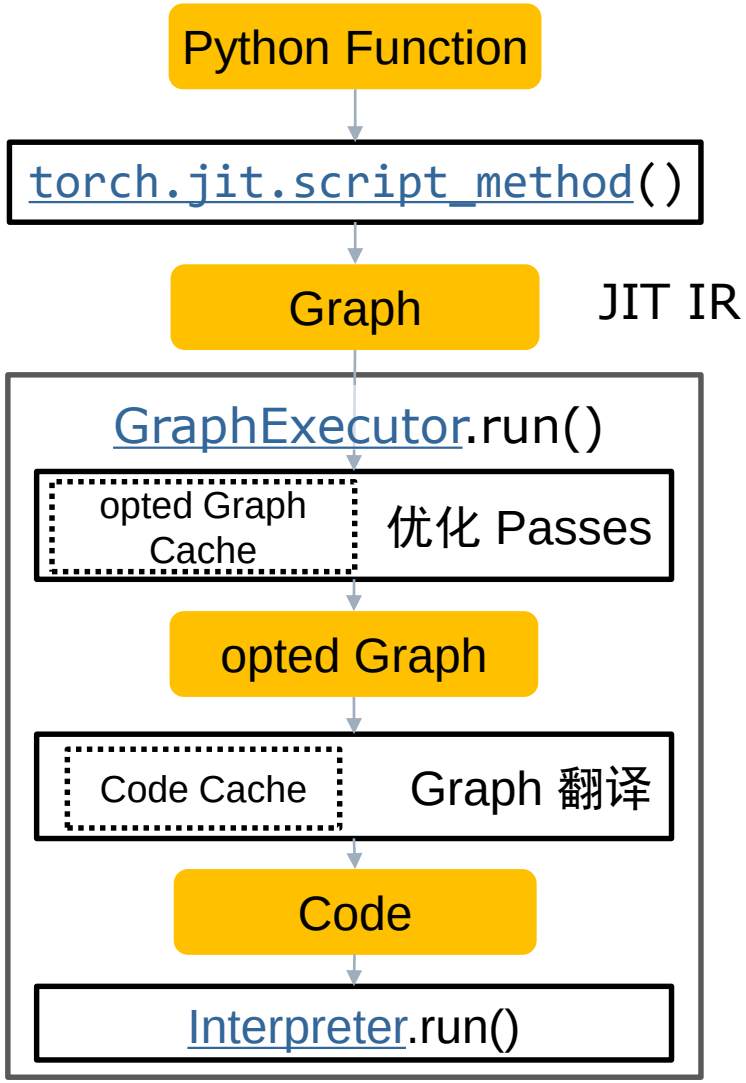
- Python转化成JIT IR(graph)
  - Torch.jit.trace

    对现有python代码和特定输入，记录在所有张量上执行的操作，生成AST再做转换
  - Torch.jit.script：

    通过Lexer和Parser将TorchScript代码➔AST，

    通过frontend直接导入AST，再对AST做语义分析生成IR

PyTorch JIT执行流程



- Python转化成JIT IR(graph)
- Graph Executor
  - 优化graph的结构
  - 维护DCG和反向图
  - 产生类汇编代码，压栈

  在很多方面都有优化，比如
  - 缓存等待命中
  - 展开小循环
  - 前向计算和传播常量
  - 窥孔优化(冗余指令删除，包括冗余的load和store指令以及死代码n控制流优化等)
- 解释器执行：基于栈，执行IR

PyTorch JIT IR

基本结构

- 最顶层module，包括parameter、Method和子模块
- Method中的Graph执行Method的功能，并实现优化
- Node表示TorchScript的一条内建功能指令
- 静态单赋值，每个值只由一个Node定义

Format

graph(%0:T_0,……,%i:T_i):
    %(i+1):T_{i+1} = namespace::name(list(Values))
    ……
    %n:T_n = namespace::name(list(Values))

%x:T 是对Value %x的类型声明
%x:T = namespace::name(input values)是一个Node

PyTorch JIT IR

- JIT IR的类型系统
  - int64_t
  - double
  - Tensor
  - Graph
  - std::string
  - 上述类型的列表(不可嵌套)

PyTorch JIT IR

- IR中的基本控制流和控制语句
  - Block中有一串Nodes，控制流Node有sub-blocks
  - Graph有graph.block()和控制流节点
  - prim::If 和 prim::Loop

prim::If

```
%y_1, ..., %y_r = prim::If(%condition)
        block0():
        %t_1, ..., %t_k = some::node(%a_value_from_outer_block)
                          -> (%t_1, ..., %t_r)
        block1():
       %f_1, ..., %f_m = some::node(%a_value_from_outer_block)
                          -> (%f_1, ..., %f_r)
```

PyTorch JIT IR

- IR中的基本控制流和控制语句
  - Block中有一串Nodes，控制流Node有sub-blocks
  - Graph有graph.block()和控制流节点
  - prim::If 和 prim::Loop

prim::If

```
@pytorch.jit.script
def f(a,b,c)
    if c:
        e = a + a
    else:
        e = b + b
    return
```

```
#D表示动态的类型
graph(%a:D,%b:D,%c:D)
  %2:D = prim::If(%c)
    block0():
      %3:int = prim::Constant[value=1]()
      %4:D = aten::add(%a,%a,%3) -> %4
    block1():
      %5:int = prim::Constant[value=1]()
      %6:D = aten::add(%b,%b,%5) -> %6
  return(%2)
```

PyTorch JIT IR

- IR中的基本控制流和控制语句
  - Block中有一串Nodes，控制流Node有sub-blocks
  - Graph有graph.block()和控制流节点
  - prim::If 和 prim::Loop

prim::Loop

```
%y_1, ..., %y_r = prim::Loop(%max_trip_count, %initial_condition,
                             %x_1, ..., %x_r)
    block0(%i, %a_1, ..., %a_r):
        %b_1, ..., %b_m = some::node(%a_value_from_outer_block, %a_1)
        %iter_condition = some::other_node(%a_2) -> (%iter_condition,
%b_1, ..., %b_r)
```

微分计算

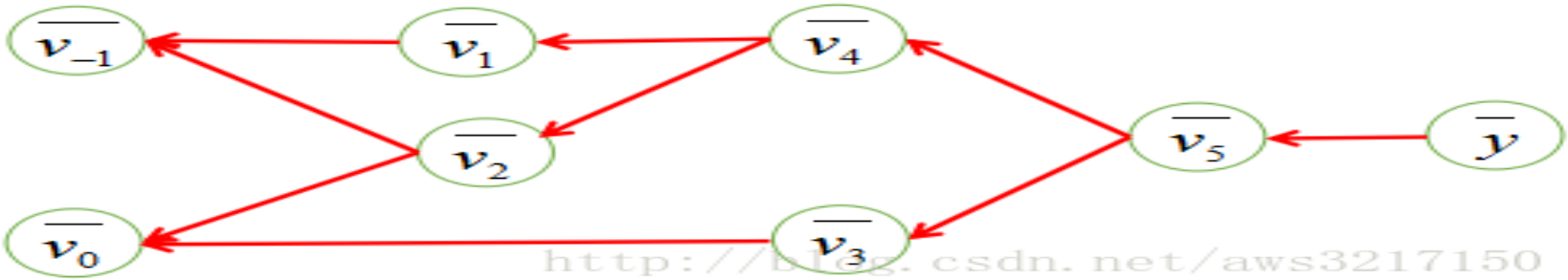
- 手动求解法(Manual Differentiation)
- 数值微分法(Numerical Differentiation)
- 符号微分法(Symbolic Differentiation)
- 自动微分法(Automatic Differentiation)

自动微分

reverse mode

图片引用链接

$f(x_1, x_2)=\ln(x_1)+ x_1x_2-\sin(x_2)$



Forward Evaluation Trace

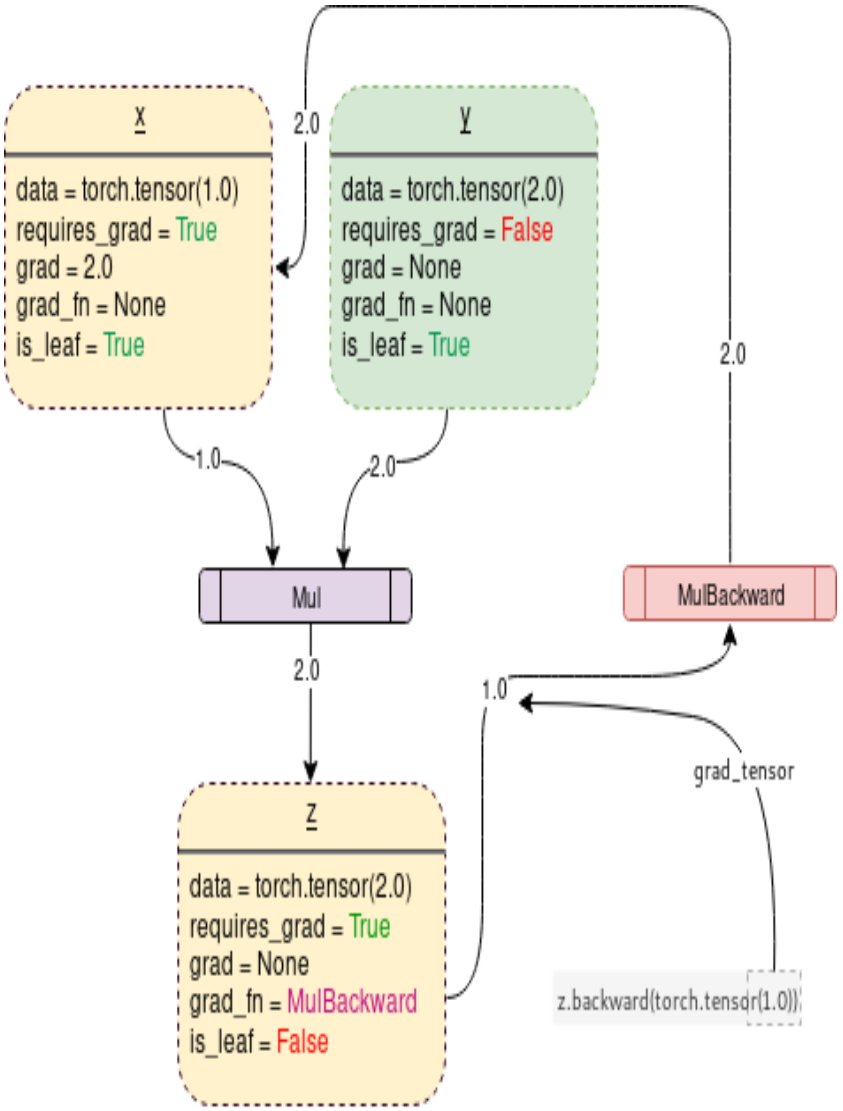
$v_{-1}=x_1$	$=2$	
$v_0=x_2$	$=5$	
$v_1=\ln v_{-1}$	$=\ln 2$	
$v_2=v_{-1}\times v_0$	$=2\times 5$	
$v_3=\sin v_0$	$=\sin 5$	
$v_4=v_1+v_2$	$=0.693+10$	
$v_5=v_4-v_3$	$=10.693+0.959$	
$y=v_5$	$=11.652$	

Reverse Adjoint Trace

$\bar{x}_1=\bar{v}_{-1}$			$=\mathbf{5.5}$
$\bar{x}_2=\bar{v}_0$			$=\mathbf{1.716}$
$\bar{v}_{-1}=\bar{v}_{-1}+\bar{v}_1\frac{\partial v_1}{\partial v_{-1}}$	$=\bar{v}_{-1}+\bar{v}_1/v_{-1}$		$=5.5$
$\bar{v}_0=\bar{v}_0+\bar{v}_2\frac{\partial v_2}{\partial v_0}$	$=\bar{v}_0+\bar{v}_2\times v_{-1}$		$=1.716$
$\bar{v}_{-1}=\bar{v}_2\frac{\partial v_2}{\partial v_{-1}}$	$=\bar{v}_2\times v_0$		$=5$
$\bar{v}_0=\bar{v}_3\frac{\partial v_3}{\partial v_0}$	$=\bar{v}_3\times\cos v_0$		$=-0.284$
$\bar{v}_2=\bar{v}_4\frac{\partial v_4}{\partial v_2}$	$=\bar{v}_4\times 1$		$=1$
$\bar{v}_1=\bar{v}_4\frac{\partial v_4}{\partial v_1}$	$=\bar{v}_4\times 1$		$=1$
$\bar{v}_3=\bar{v}_5\frac{\partial v_5}{\partial v_3}$	$=\bar{v}_5\times(-1)$		$=-1$
$\bar{v}_4=\bar{v}_5\frac{\partial v_5}{\partial v_4}$	$=\bar{v}_5\times 1$		$=1$
$\bar{v}_5=\bar{y}$			$=1$

PyTorch微分机制

- 基于动态计算图、自动微分reverse mode
- 使用PyTorch的微分
  - 定义forward方法
  - 记录执行的操作，在后端产生DCG和后向图
  - 调用backward方法，沿后向图计算梯度直到叶节点
- Backward方法



PyTorch及其JIT IR的总结

- 优点
  - 使用方便
    - TorchScript是PyThon语言的子集，便于开发
    - 执行中建立动态计算图，调试简单直观
- 缺点
  - 目前暂不支持递归

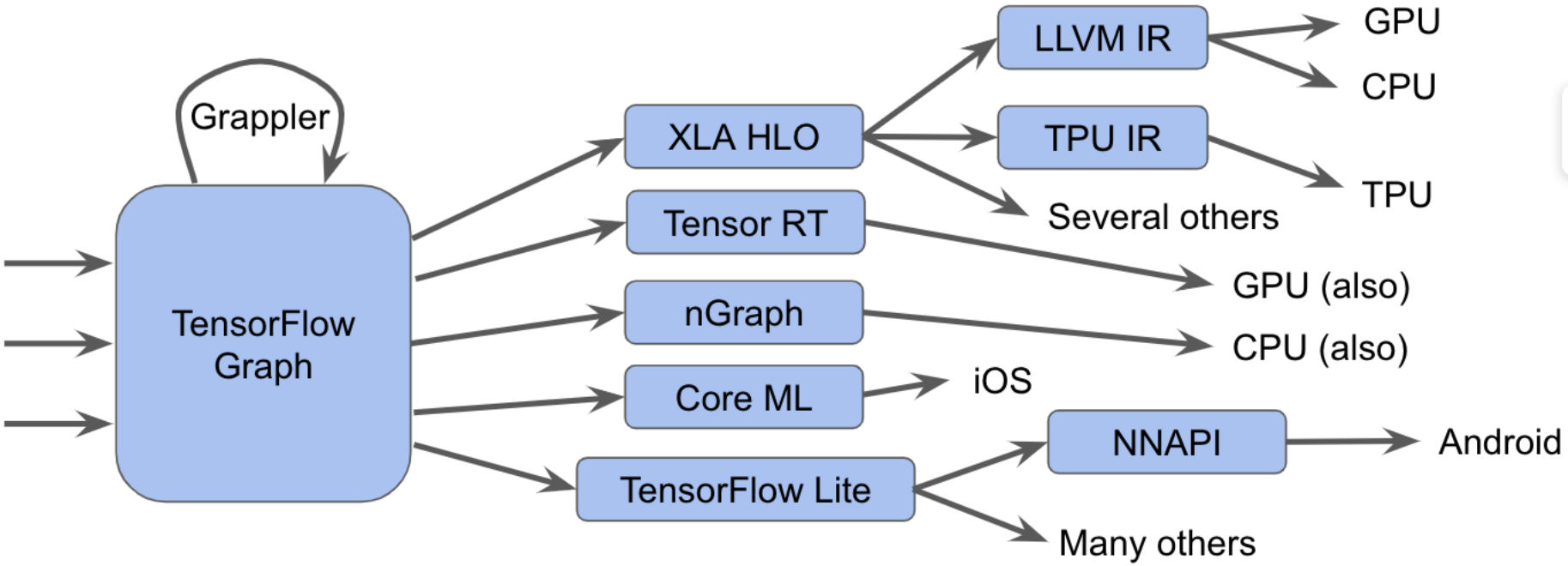
中国科学技术大学
University of Science and Technology of China

TensorFlow MLIR

Motivation

- 我们可以怎样运行一张TensorFlow图？
  - 转换成XLA高级优化表示（XLAHLO），然后再转换成LLVM IR或者TPU IR
  - 将图转化为 TensorFlow Lite 格式
  - 将图转化为 TensorRT 、 nGraph或另一种适合特定硬件指令集的编译器格式
  - ……

实际情况更复杂


Grappler
TensorFlow Graph
XLA HLO
Tensor RT
nGraph
Core ML
TensorFlow Lite
LLVM IR
TPU IR
Several others
GPU
CPU
TPU
GPU (also)
CPU (also)
iOS
NNAPI
Android
Many others

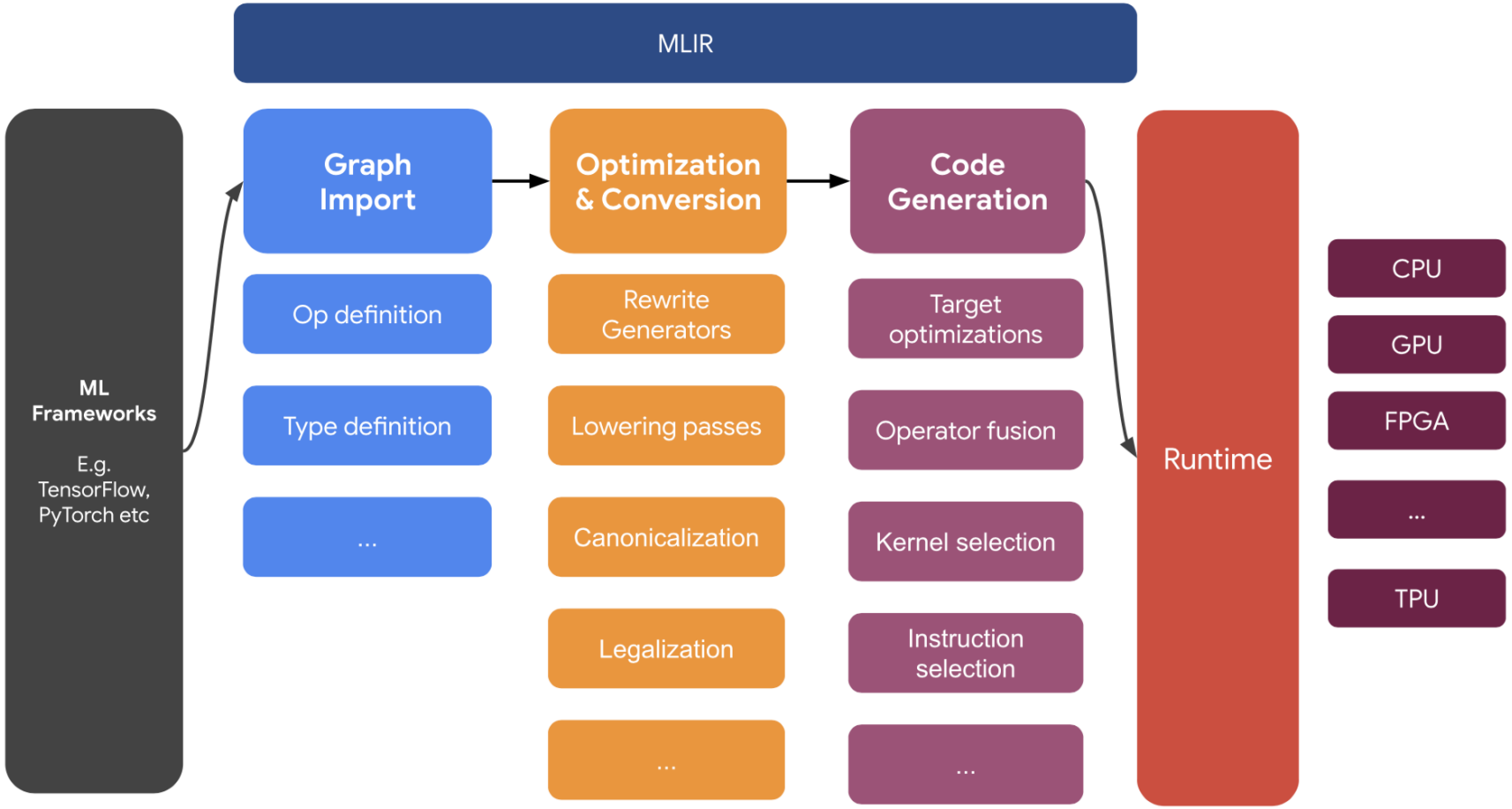
导致的问题

- 系统间的边界处总是会产生令人困惑的错误
- 若需要构建新的软硬件堆栈生成器，则必须为每个新路径重新构建优化与转换传递
- 参考文章
  - *A new intermediate representation and compiler framework, 2019.4.9*

MLIR

- **MLIR**：**M**ulti-**L**evel **I**ntermediate **R**epresentation
  - 用于现代优化编译器的灵活基础架构
  - 由IR规范和代码工具包组成，该工具包可对该表示进行转换
  - 允许在同一编译单元中表示、分析、转换成多级别抽象图，包括
    - TensorFlow操作
    - 嵌套的多面体Polyhedral 循环
    - LLVM指令
    - 固定的硬件操作和类型

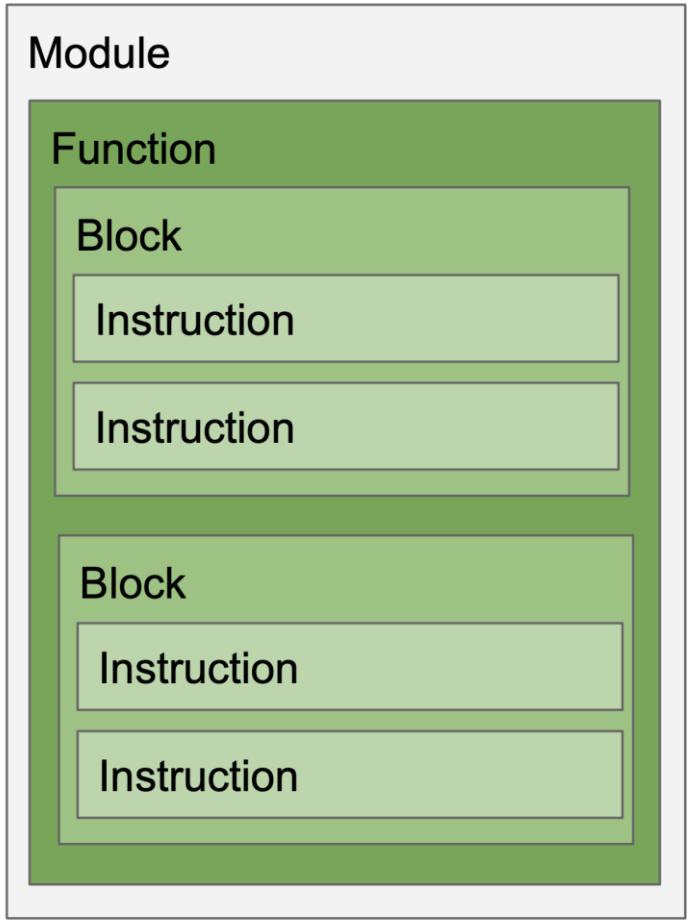
MLIR


MLIR
ML Frameworks
E.g. TensorFlow, PyTorch etc
Graph Import
Op definition
Type definition
...
Optimization & Conversion
Rewrite Generators
Lowering passes
Canonicalization
Legalization
...
Code Generation
Target optimizations
Operator fusion
Kernel selection
Instruction selection
...
Runtime
CPU
GPU
FPGA
...
TPU

与LLVM的相似点

- SSA, typed, three address
- Module/Function/Block
- Instruction structure
- Round trippable textual for
- Syntactically similar:

```
func @testFunction(%arg0: i32) {
  %x = call @thingToCall(%arg0) : (i32) -> i32
  br ^bb1
^bb1:
  %y = addi %x, %x : i32
  return %y : i32
}
```



与LLVM的不同点

基本块的传递

```
func @search_body(%A: memref<?x?xi32>, %S: memref<?xi32>, %key: i32) {
  %nj = dim %A, 1 : memref<?x?xi32>
  br ^bb1(0)

^bb1(%j: i32)
  %p1 = cmpi "lt", %j, %nj : i32
  cond_br %p1, ^bb2, ^bb5

^bb2:
  %v = affine.load %A[%i, %j] : memref<?x?xi32>
  %p2 = cmpi "eq", %v, %key : i32
  cond_br %p2, ^bb3(%j), ^bb4

^bb3(%j: i32)
  affine.store %j, %S[%i] : memref<?xi32>
  br ^bb5

^bb4:
  %jinc = addi %j, 1 : i32
  br ^bb1(%jinc)

^bb5:
  return
}
```

Dialects

- 与MLIR生态系统互动和扩展的机制
- 允许定义新操作（operation）以及属性和类型
- 每个方言都有一个唯一的名称空间
  - 每一个属性、操作、类型都有一个dialect的前缀
  - E.g. GPU dialect, LLVM dialect, Vector dialect

```
module attributes {gpu.container_module} {

  // This module creates a separate compilation unit for the GPU compiler.
  module @kernels attributes {gpu.kernel_module} {
    func @kernel_1(%arg0 : f32, %arg1 : !llvm<"float*">)
        attributes { nvvm.kernel = true } {

      // Operations that produce block/thread IDs and dimensions are injected when
      // outlining the `gpu.launch` body to a function called by `gpu.launch_func`.
      %tIdX = "gpu.thread_id"() {dimension = "x"} : () -> (index)
      %tIdY = "gpu.thread_id"() {dimension = "y"} : () -> (index)
      %tIdZ = "gpu.thread_id"() {dimension = "z"} : () -> (index)
```

Dialects

- MLIR允许多个方言，即使是主树之外的方言，也可以在一个模块中共存。
- MLIR提供了一个具体框架MLIR(framework) 在dialects中之间和之内转换

中国科学技术大学
University of Science and Technology of China

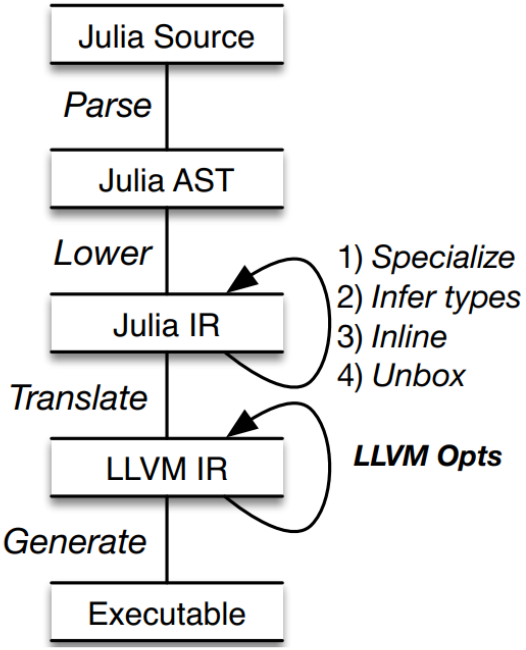
Zygote.jl & Julia IR

- Julia 编译流程
- Julia IR
- Zygote.jl 微分机制

Julia编译流程-oopsla2018

编译流程

- julia-parser.scm将Julia代码解析并转换
- 在conpiler/typeinfer.jl中实现类型推断
- codegen.cpp将Julia AST转化为本机机器代码



oopsla2018 Julia fig.8

Julia IR

- 以pow函数为例
  - 以基本块划分
  - goto-based的控制流

```
1  @code_lowered pow(2,6)
```

```
1  function pow(x,n)
2      r=1
3      while n>0
4          n-=1
5          r*=x
6      end
7      return r
8  end
```

```
CodeInfo(
1 ─       n@_5 = n@_3
└──       r = 1
2 ┄ %3 = n@_5 > 0
└──       goto #4 if not %3
3 ─       n@_5 = n@_5 - 1
│         r = r * x
└──       goto #2
4 ─       return r
)
```

Zygote.jl微分机制-CoRR2018

- 传统的微分机制
  - 基于跟踪——计算图（动态 vs 静态）

- Julia的微分机制
  - 基于编译手段
    - J函数与回调函数——前向计算求值和反向微分
    - 原始代码和伴随代码——微分过程代码的产生
      - 如何生成伴随代码？
      - 如何在反向重放控制流时正确引用原始代码的值？
    - 针对语言特性实现微分

Zygote.jl微分机制-CoRR2018

J函数与回调函数

1.高阶函数J：

- 计算函数f的结果并产生回调函数

$$y, \mathcal{B}_y = \mathcal{J}(f, x_1, x_2, ...)$$

2. 回调函数B_y

- 接收结果相对于y的梯度，返回结果相对x的梯度

$$\bar{x} = \frac{\partial l}{\partial x} = \frac{\partial l}{\partial y}\frac{\partial y}{\partial x} = \mathcal{B}_y(\bar{y})$$

Zygote.jl微分机制-CoRR2018

原始代码和伴随代码

- 如何生成伴随代码？
  - 安插额外Phi结点记录原始代码控制流
- 如何在反向重放控制流时正确引用原始代码的值？
  - alpha结点入栈
  - 检查点技术

Zygote.jl微分机制-CoRR2018

举例

$$f(x) = \begin{cases} x & x > 0 \\ 0.01x & \text{otherwise} \end{cases}$$

原始代码

block #1:
  $\%1 \leftarrow x > 0$
  goto #3 **if** $\%1$
block #2:
  $\%2 \leftarrow 0.01x$
block #3:
  $\%3 \leftarrow \phi(\#1 \rightarrow x, \#2 \rightarrow \%2)$
  return $\%3$

伴随代码

block #1:
  goto #3 **if not** $\%4$
block #2:
  $\_, \%1 \leftarrow \%3(\bar{y})$
  goto #3
block #3:
  $\%2 \leftarrow \phi(\#1 \rightarrow \bar{y}, \#2 \rightarrow \%1)$
  return $\%2$

原始代码+额外节点

block #1:
  $\%1 \leftarrow x > 0$
  goto #3 **if** $\%1$
block #2:
  $\%2, \%3 \leftarrow \mathcal{J}(\times, 0.01, x)$
block #3:
  $\%4 \leftarrow \phi(\#1 \rightarrow false, \#2 \rightarrow true)$
  $\%5 \leftarrow \phi(\#1 \rightarrow x, \#2 \rightarrow \%2)$
  return $\%5$

Zygote.jl微分机制-CoRR2018

- 如何对语言特性实现微分？
  - 两种基本数据结构可以微分
    - Cons cell （二元组）
    - Box （可变元素）
  - 这两种基本数据结构的复合可以微分
    - 栈
    - 链表
    - …

Julia自动微分总结

- 优点
  - 自动微分所用IR就是Julia IR，可以直接使用Julia的JIT，无需额外实现新的JIT
  - 可以复用一些比较传统的编译优化手段
  - 支持递归、循环、分支、一些数据结构的微分
- 缺点（个人猜测）
  - 实现较为复杂

中国科学技术大学

University of Science and Technology of China

总结

三种形式的相同点与不同点

- 相同点
  - 三种IR的基本结构类似
- 不同点
  - MLIR优化机制与其余两种不同
  - Julia使用的是自己语言的JIT，另外两种使用的是框架的JIT
  - IR执行方式不同，Pytorch是基于栈执行，Julia直接使用自己语言的JIT执行

中国科学技术大学

University of Science and Technology of China

谢谢