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

人工智能模型的编译

《编译原理和技术(H)》

张昱

0551-63603804，yuzhang@ustc.edu.cn

中国科学技术大学

计算机科学与技术学院

大模型与AI系统基础设施

大模型

- 训练框架

  分布式并行、集群通信等

  算力+存储+网络

- 推理框架

  如 vLLM、SGLang、FastTransformer

  云端大算力+端侧轻算力

  成本可控下低延迟、高吞吐

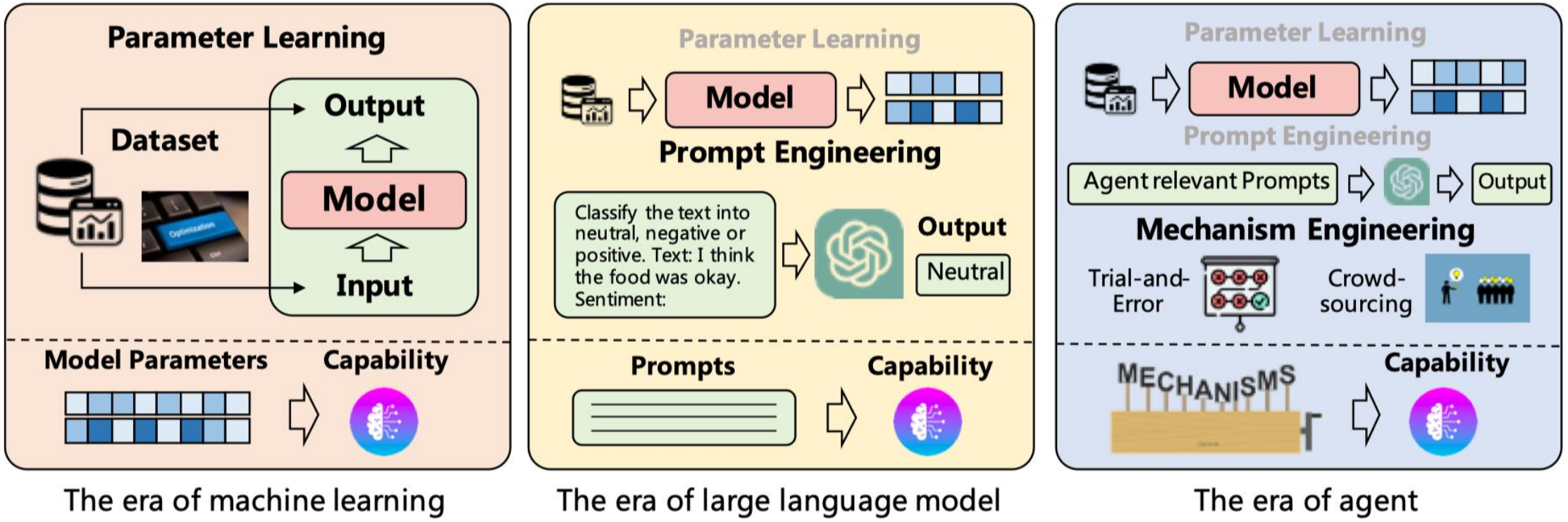
- AI编程框架
- 编译器
- AI芯片组成AI集群

大模型与智能体

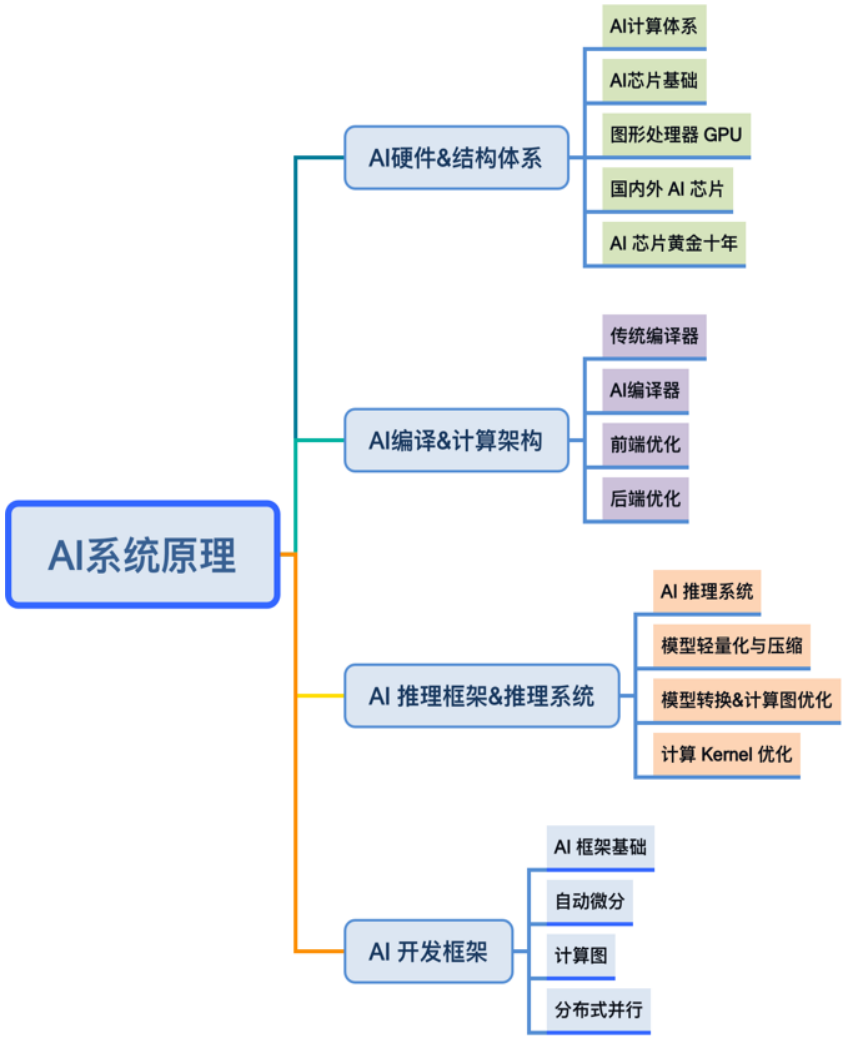
- 大语言模型LLM：响应用户的查询指令，实现一些生成任务
- 大动作模型LAM(Large-Action Models/Large-Agent Models)

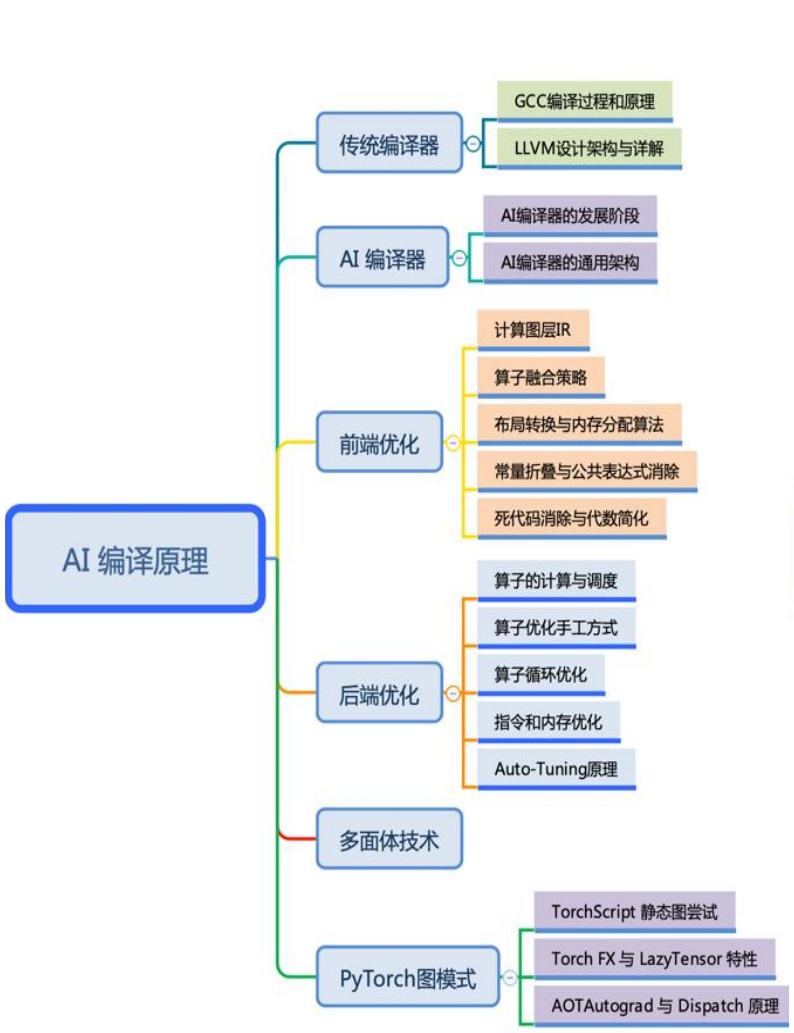
以LLM为Agent的中心，将复杂任务分解，在每个子步骤实现自主决策和执行

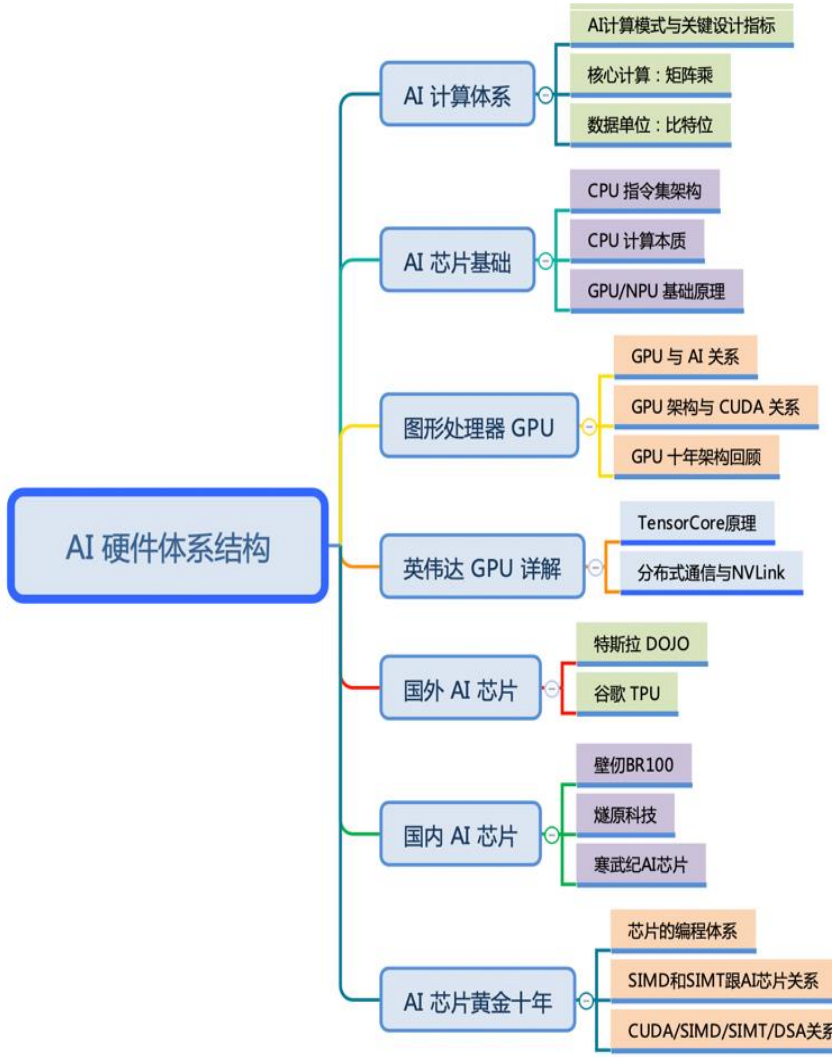
Agent = LLM + Planning 计划+ Tool use 执行 + Feedback 纠正偏差



AI系统与编译

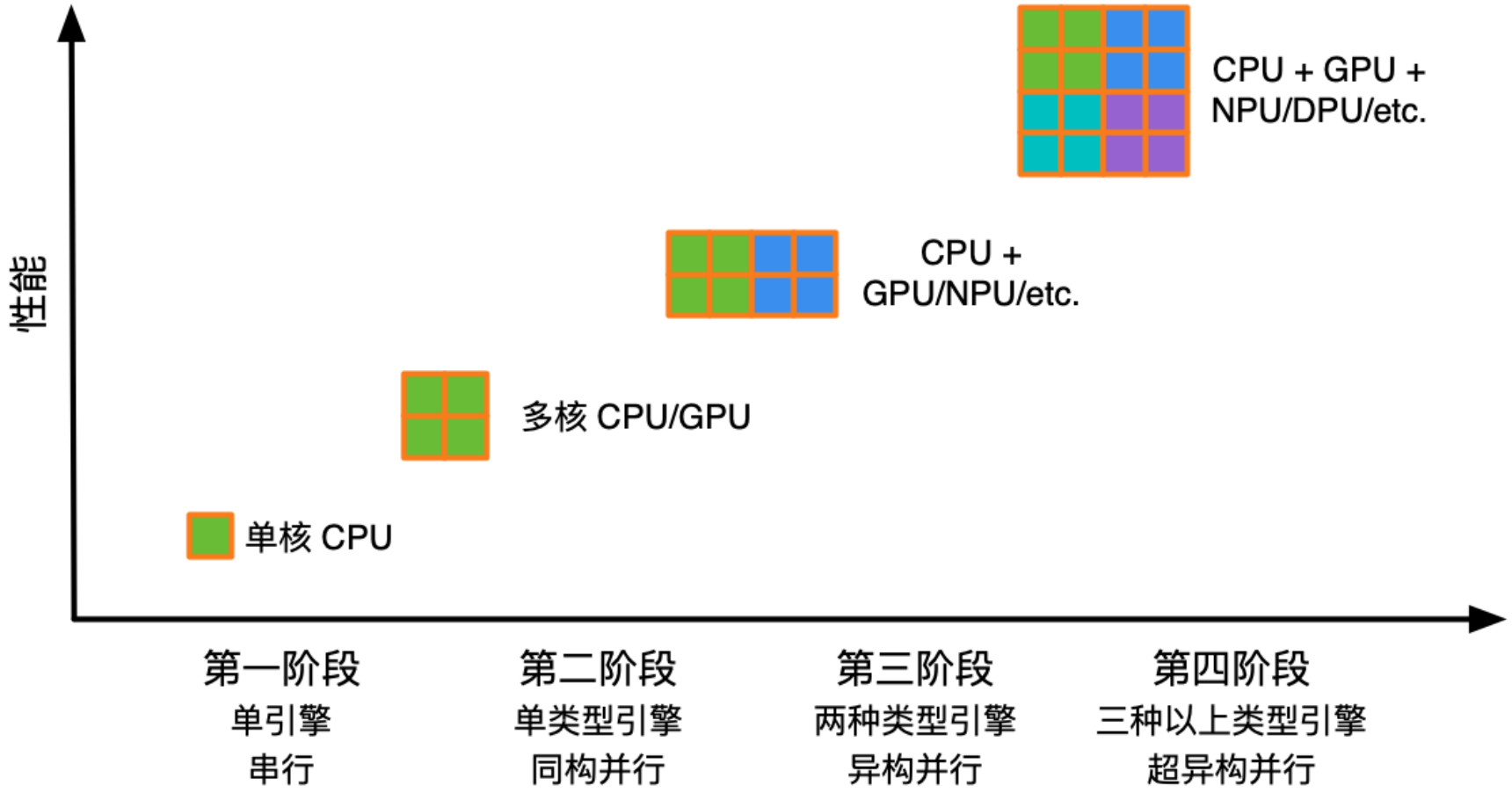

AI系统原理
AI硬件&结构体系
AI计算体系
AI芯片基础
图形处理器 GPU
国内外 AI 芯片
AI 芯片黄金十年
AI编译&计算架构
传统编译器
AI编译器
前端优化
后端优化
AI 推理框架&推理系统
AI 推理系统
模型轻量化与压缩
模型转换&计算图优化
计算 Kernel 优化
AI 开发框架
AI 框架基础
自动微分
计算图
分布式并行


AI 编译原理
传统编译器
GCC编译过程和原理
LLVM设计架构与详解
AI 编译器
AI编译器的发展阶段
AI编译器的通用架构
前端优化
计算图层IR
算子融合策略
布局转换与内存分配算法
常量折叠与公共表达式消除
死代码消除与代数简化
后端优化
算子的计算与调度
算子优化手工方式
算子循环优化
指令和内存优化
Auto-Tuning原理
多面体技术
PyTorch图模式
TorchScript 静态图尝试
Torch FX 与 LazyTensor 特性
AOTAutograd 与 Dispatch 原理


AI 硬件体系结构
AI 计算体系
AI计算模式与关键设计指标
核心计算：矩阵乘
数据单位：比特位
AI 芯片基础
CPU 指令集架构
CPU 计算本质
GPU/NPU 基础原理
图形处理器 GPU
GPU 与 AI 关系
GPU 架构与 CUDA 关系
GPU 十年架构回顾
英伟达 GPU 详解
TensorCore原理
分布式通信与NVLink
国外 AI 芯片
特斯拉 DOJO
谷歌 TPU
国内 AI 芯片
壁仞BR100
燧原科技
寒武纪AI芯片
AI 芯片黄金十年
芯片的编程体系
SIMD和SIMT跟AI芯片关系
CUDA/SIMD/SIMT/DSA关系

硬件

- CPU
- GPU
- AI专用处理器
  - 华为昇腾NPU
  - 谷歌TPU
  - 特斯拉DOJO
  - ……



算力单位

OPS

- **OPS**(Operations Per Second)，**1 TOPS** 代表处理器每秒进行一万亿次(10^{12})计算
- **OPS/W** 每瓦特运算性能，**TOPS/W** 评价处理器在1W 功耗下运算能力的性能指标

FLOPs

- **浮点运算次数**(Floating Point Operations)用来衡量模型计算复杂度，常用作神经网络模型速度的间接衡量标准。对于卷积层而言，$FLOPs = 2 \cdot H \cdot W \cdot C_{in} \cdot K \cdot K \cdot C_{out}$

(H,W:输出特征图的高度和宽度；Cin,Cout输入或输出通道数; K卷积核的尺寸)

MACs

- **乘加累积操作**Multiply–Accumulate Operations，**1MACs** 包含一个乘法操作与一个加法操作，~2FLOPs，通常MACs与FLOPs存在一个2倍的关系。

MAC

- **内存占用量**(Memory Access Cost)，用来评价模型在运行时的内存占用情况。 1x1 卷积 FLOPs为 $2 \cdot H \cdot W \cdot C_{in} \cdot C_{out}$，其对应MAC为： $H \cdot W \cdot (C_{in} + C_{out}) + (C_{in} * C_{out})$

AI芯片关键指标

1. 精度 Accuracy
   - 计算精度 (FP32/FP16 etc.)
   - 模型结果精度 (ImageNet 78%)
2. 吞吐量 Throughput
   - 高维张量处理 (high dimension tensor)
   - 实时性能 (30 fps or 20 tokens)
3. 时延 Latency
   - 交互应用程序 (TTA)
4. 能耗 Energy
   - IOT 设备有限的电池容量
   - 数据中心液冷等大能耗
5. 系统价格 System Cost
   - 硬件自身的价格 $$$
   - 系统集成上下游全栈等成本
6. 易用性 Flexibility
   - 衡量开发效率和开发难度

AI加速器的关键设计点

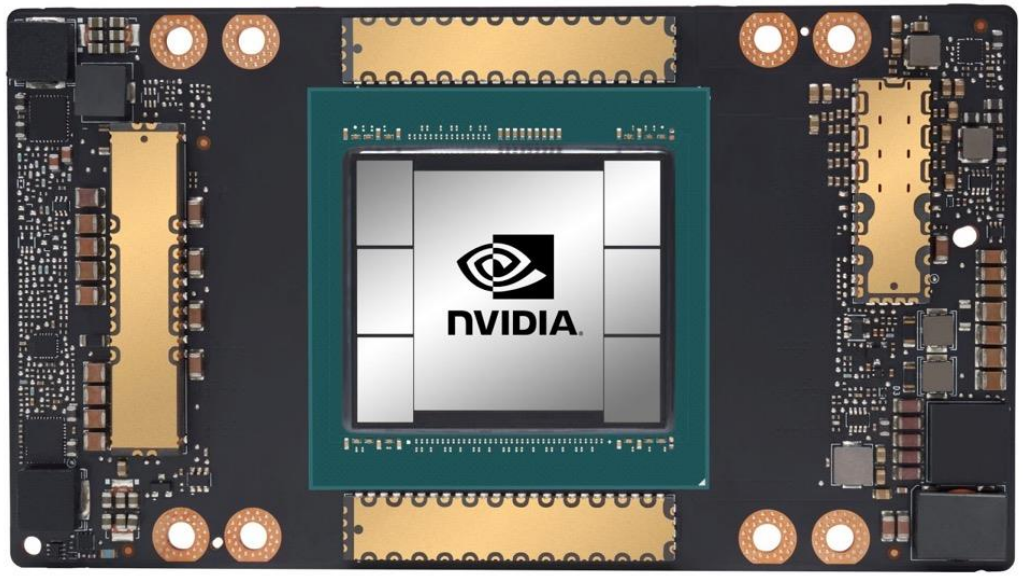
提升吞吐量、降低时延

低时延与batch size之间权衡

AI加速器

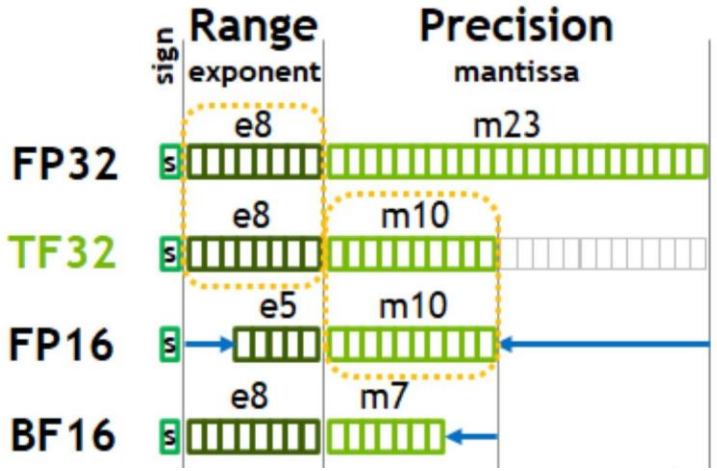
支持 8-bit 推理 & 16-bit float 训练的产品

华为昇腾 910



NVIDIA A100

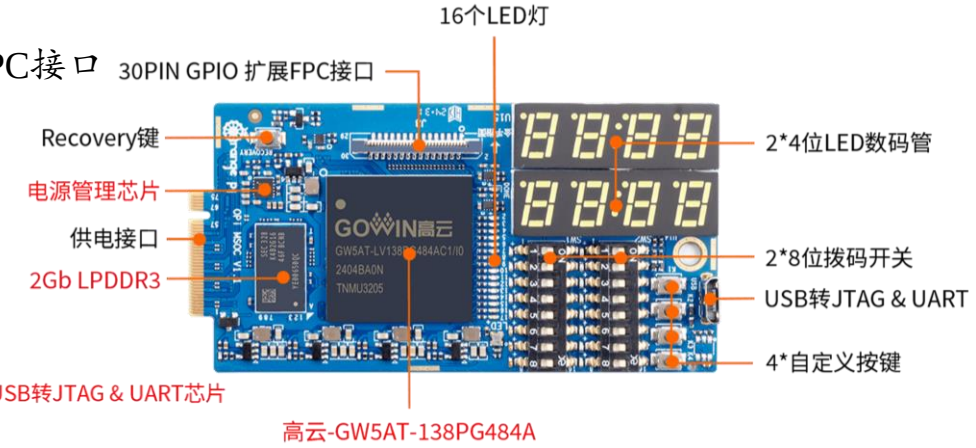
- **训练**：使用 FP16、BF16、TF32；
- **推理**：CV 任务以 int8 为主，
  NLP 以 FP16为主，大模型 int8/FP16 混合

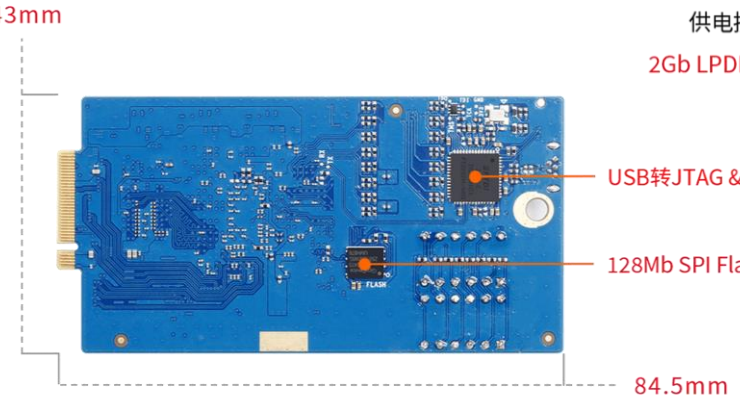


香橙派鲲鹏开发板(含FPGA)

- **OrangePi Kunpeng Pro (16G)+FPGA (OPi MSOC)**
  - **OS：openEuler 22.03**
  - **存储：** 板载 32MB 的 SPI Flash; SATA/NVME SSD(M.2接口2280)
    eMMC插座：可外接eMMC模块; eMMC5.1 HS400; TF插槽
  - **接口：** 40Pin 功能扩展接口，支持GPIO、UART、I2C、SPI、I2S、PWM
  - **FPGA：** 高云GW5AT-138PG484A
    内存LPDDR3 :2Gb ； 存储：SPI Flash: 128Mb；30PIN GPIO 扩展FPC接口
- 资料 https://www.hikunpeng.com/developer/devboard
  http://www.orangepi.cn/html/hardWare/computerAndMicrocontrollers/details/Orange-Pi-MSOC.html
  - 用户手册、环境手册
  - 系列在线课程

OrangePi Kunpeng Pro（教学场景）
107*68mm,82g





香橙派昇腾开发板(310B)

OrangePi AIPro（20T）
115.23mm* 83.26m* 1.6mm, 82g

OrangePi Alpro(20T)

- **OS：Ubuntu 22.04.3 LTS、openEuler 22.03、内核 Linux 5.10.0+**
- **内存**：LPDDR4X：24GB，速率：4266Mbps
  支持eMMC模块：32GB/64GB/256GB、SATA/NVME SSD（M.2接口2280）、TF插槽、SPI Flash: 32MB
- **AI算力**：20 TOPS（INT8）
- **CPU算力**：
  1 DaVinciV300 AI core * 1.224 GHz；4 TAISHANV200M core * 1.6GHz
- **接口**：40Pin 功能扩展接口，支持GPIO、UART、I2C、SPI、I2S、PWM

资料 https://www.hiascend.com/developer/devboard

- 用户手册、环境手册
- 系列在线课程

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

AI编译及实例分析

- AI编译器的发展
- Pytorch+Triton+LLVM

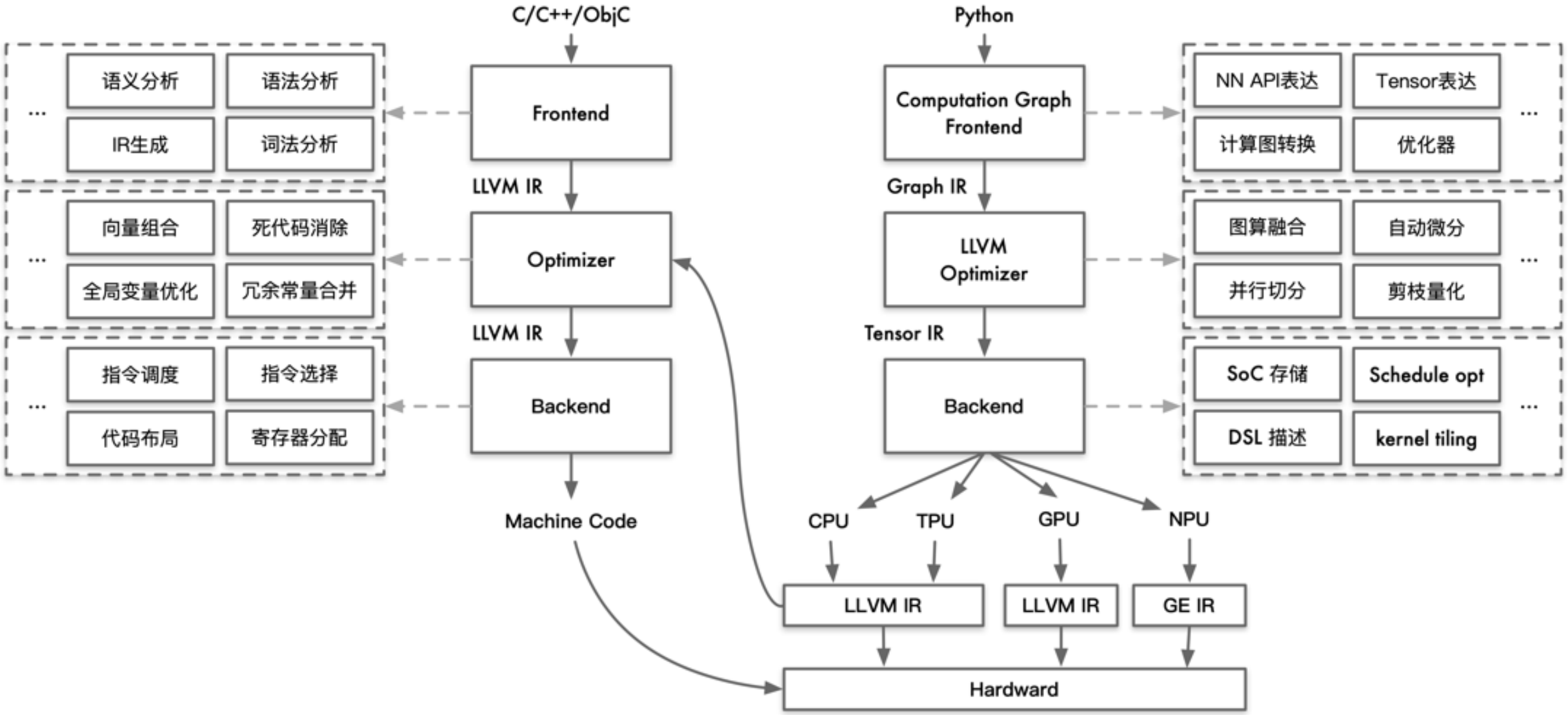
AI编译器的发展阶段 朴素 → 专用 → 通用

Naive	Specific	Universal
1. 以计算图和算子抽象为主 2. 计算图中采用部分编译器技术	1. 类PyTorch的Python原生表达，静态化转换 2. AI专用编译器架构，打开图算边界进行融合优化	1. 图算统一表达，实现融合优化 2. 算子自动生成，降低开发门槛 3. 针对神经网络，泛化优化能力 4. 模块化表示组合，提升可用性
Stage I	Stage II	Stage III
Caffe TensorFlow		
Cons 1. API构图，易用性差 2. 大量DSA异构芯片对性能挑战 3. 算子边界确定无法充分发挥性能		Cons 1. 图算表达分开表达 2. 神经网络的功能泛化 3. 算子实现Schedule等缺乏自动化

推理场景：性能优化、资源利用、模型压缩、硬件兼容性

训练场景：代码生成、梯度计算、并行计算、内存管理

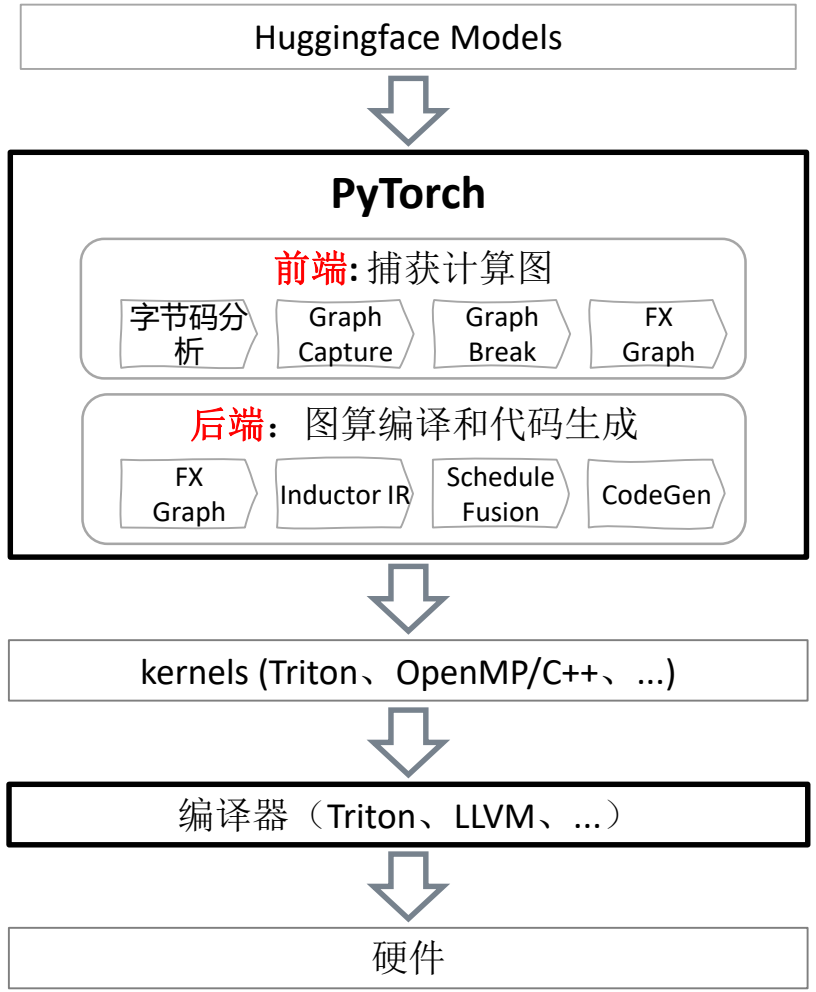
AI编译器：站在传统编译器肩膀之上



深度学习模型编译软件栈

核心思想：①Python 代码➔计算图，
②对图进行优化和代码生成：图-算子两层，
③将生成的代码编译为可执行程序。
典型软件栈例如：

- **PyTorch前端捕获计算图**
  - **输入：**用户编写的动态的 Python 代码
  - **输出：**静态、可供编译器分析的计算图
- **PyTorch后端负责优化和代码生成**
  - **输入：**静态计算图
  - **输出：**生成的 Triton 核函数代码
- **底层编译器将核函数编译成可执行程序**



中国科学技术大学

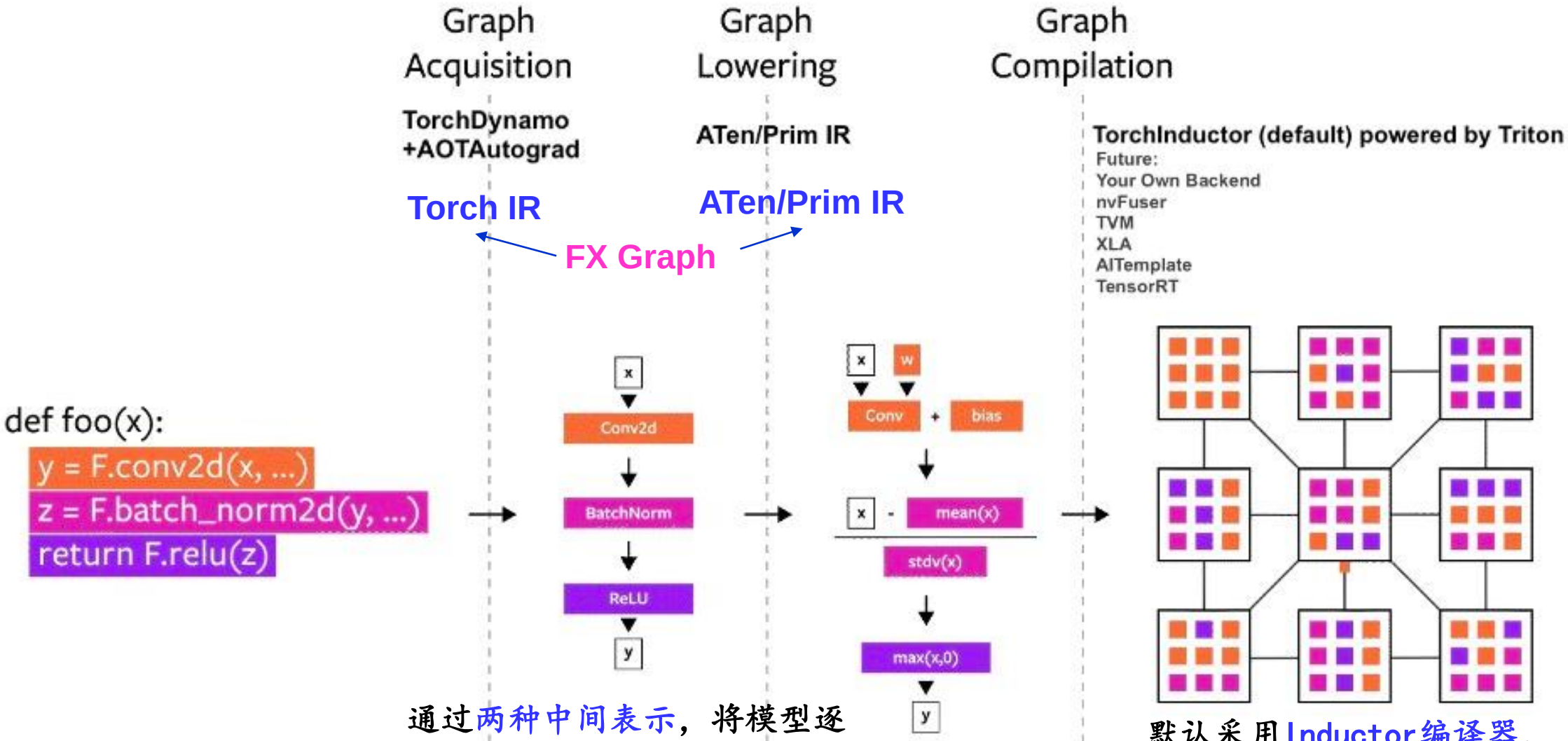
University of Science and Technology of China

X.1 PyTorch 简介

PyTorch 2.0 下一代深度学习框架[https://pytorch.org/]

- Meta AI, 从2023年3月起
- 核心特性: torch.compile()，一行代码实现加速
  - Faster: 无需修改大量代码，通过编译技术显著提升模型训练和推理速度
  - More Pythonic: 在提供高性能的同时，完整保留PyTorch 的动态性和灵活性
- 资源
  - doc, github；Video：PyTorch Conference 2022, Dec 13, 2022
  - 论文：PyTorch 1.0[NeurIPS 2019]、PyTorch 2.0[ASPLOS2024]

PyTorch编译过程



通过两种中间表示，将模型逐级下降为统一的计算图表示

默认采用Inductor编译器，为不同硬件生成专用的核函数

X.1.1 TorchDynamo

- PyTorch 2.0[ASPLOS2024]: PyTorch 2.0 原理

第一层：Dynamo

针对前向(Forward)计算捕获计算图

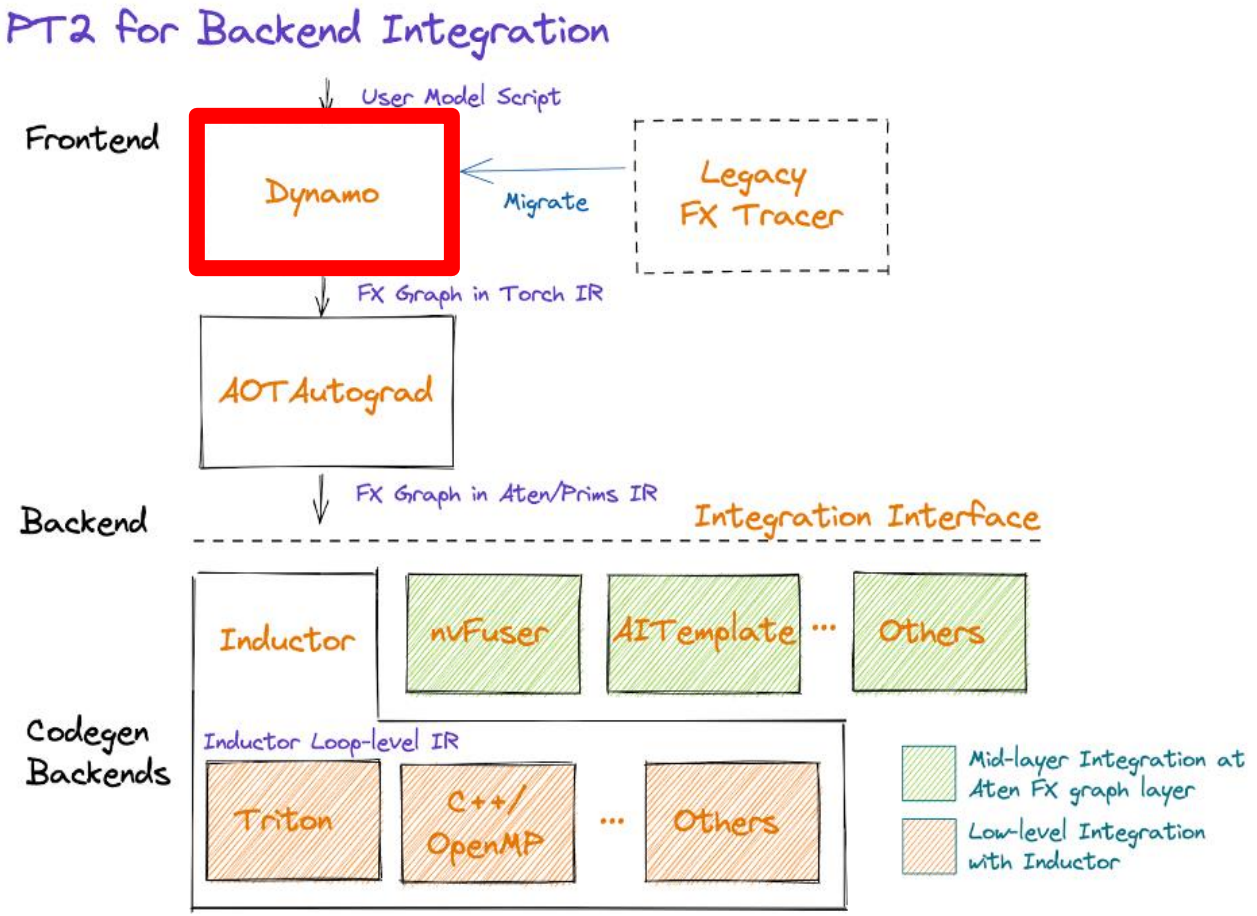
- **Python 字节码的捕获与转换**
  - 不能捕获的计算图，用eager mode运行
  - 能捕获的，dynamo 处理后交给后端继续编译

```
def f(x):
    a=x*2
    b=a + torch.from_numpy(np.randn(5))
    if b.sum()>0:
        return b.sin().sin()
    else:
        return b.cos().cos()
```

子图 1

子图 2

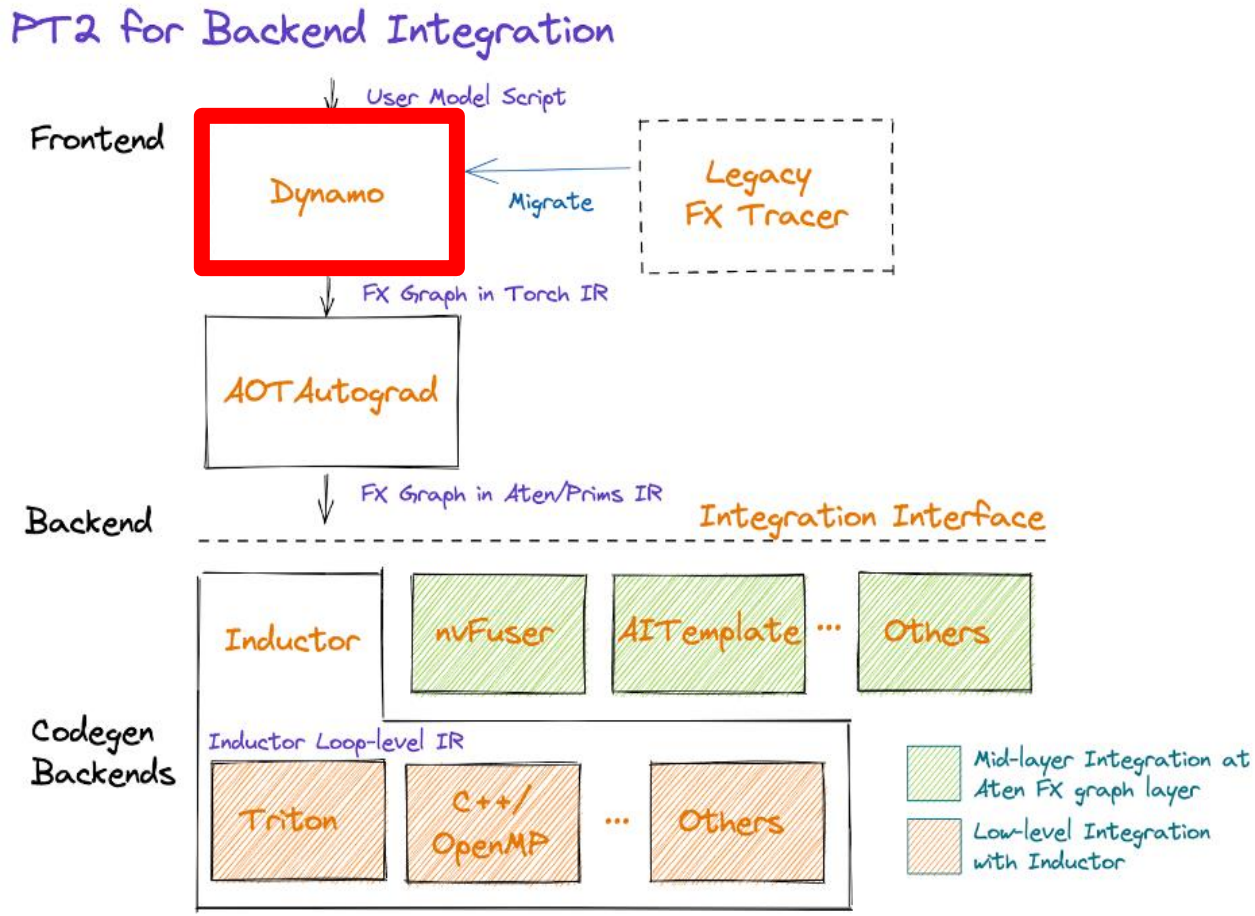
子图 3



第一层：Dynamo

针对前向(Forward)计算捕获计算图

- Python 字节码的捕获与转换
  - 不能捕获的计算图，用eager mode运行
  - 能捕获的，dynamo 处理后交给后端继续编译
- 生成 Torch IR (FX Graph)



Torch IR (FX Graph)

图表示 IR：表示图结构和执行图转换

IR 的核心数据结构

- fx.Graph 静态单赋值(SSA)，包含一组 Node
- fx.Node 表示一个计算步骤，包含属性 op、target、 args、kwargs，数据依赖通过边（即Node的args 和 kwargs）显式表示

支持动态形状

- 通过符号数表示Dynamo捕获的动态维度
  符号数：SymInt，SymFloat，…
  符号计算函数：sym_max，sym_sum，…

fx.Graph 封装成可调用的 Python Code

- 封装为 fx.GraphModule，通过 forward 方法被调用

```
class GraphModule(torch.nn.Module):
    def forward(self, ... ):
        ...
        input_1 = _nn.linear(l_args_0_, ...)
        input_2 = nn.functional.layer_norm(input_1,
           (128,), ...)
        input_3 = nn.functional.silu(input_2, inplace
           = False)
        # 剩下的两层网络
        ...
        return (input_9,)
```

Torch IR 对应的 Python 代码示例

op	target	args	kwargs
call_module	nn.functional.silu	(input_2,)	{inplace: False}

Node input_3 的属性

[Arxiv2112.08429] torch.fx: Practical Program Capture and Transformation for Deep Learning In Python

Python 字节码的捕获与转换

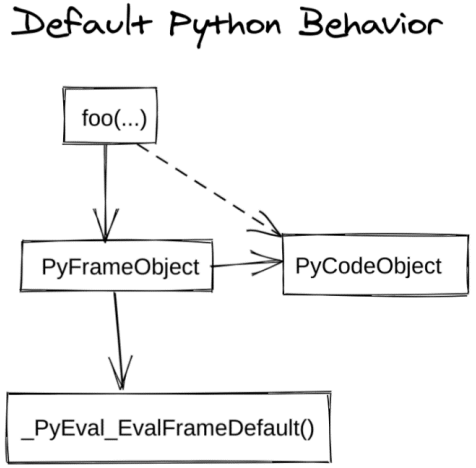
拦截和分析 Python 字节码

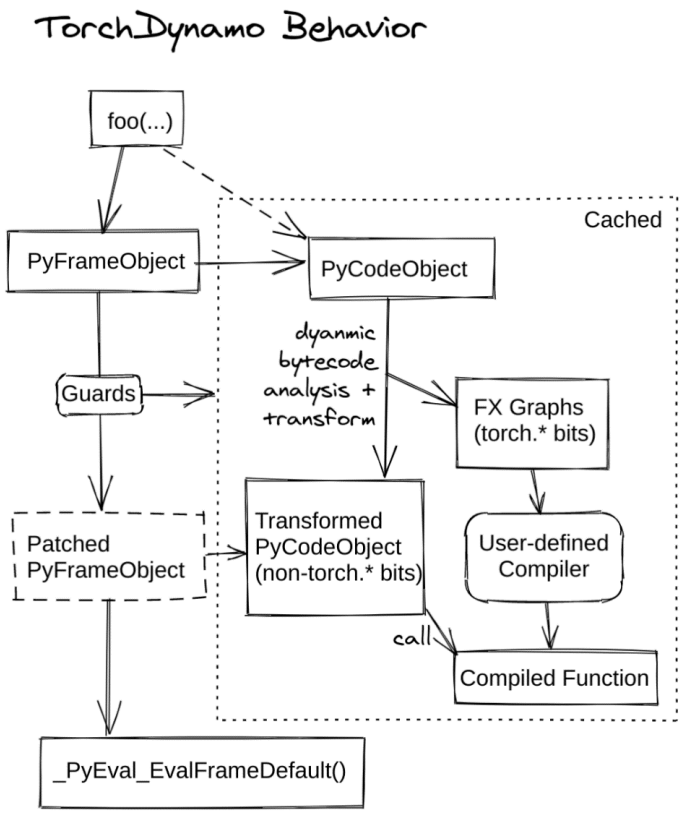
拦截

`torch.compile`返回的函数 `compiled`

首次执行时，Dynamo 会在 Python 解释器**执行其字节码(_PyEval_EvalFrameDefault)**之前，拦截下来，不实际执行

```python
n, h = 32, 128
repeats = 3
layers = OrderedDict()
for i in range(repeats):
    layers[f"fc_{i}"] = nn.Linear(h, h)
    layers[f"ln_{i}"] = nn.LayerNorm(h)
    layers[f"silu_{i}"] = nn.SiLU()
model = nn.Sequential(layers).cuda().half()
x = torch.randn((n, h), device="cuda", dtype=torch.float16)
compiled = torch.compile(model, mode="reduce-overhead")
with torch.no_grad():
    y = compiled(x)
```





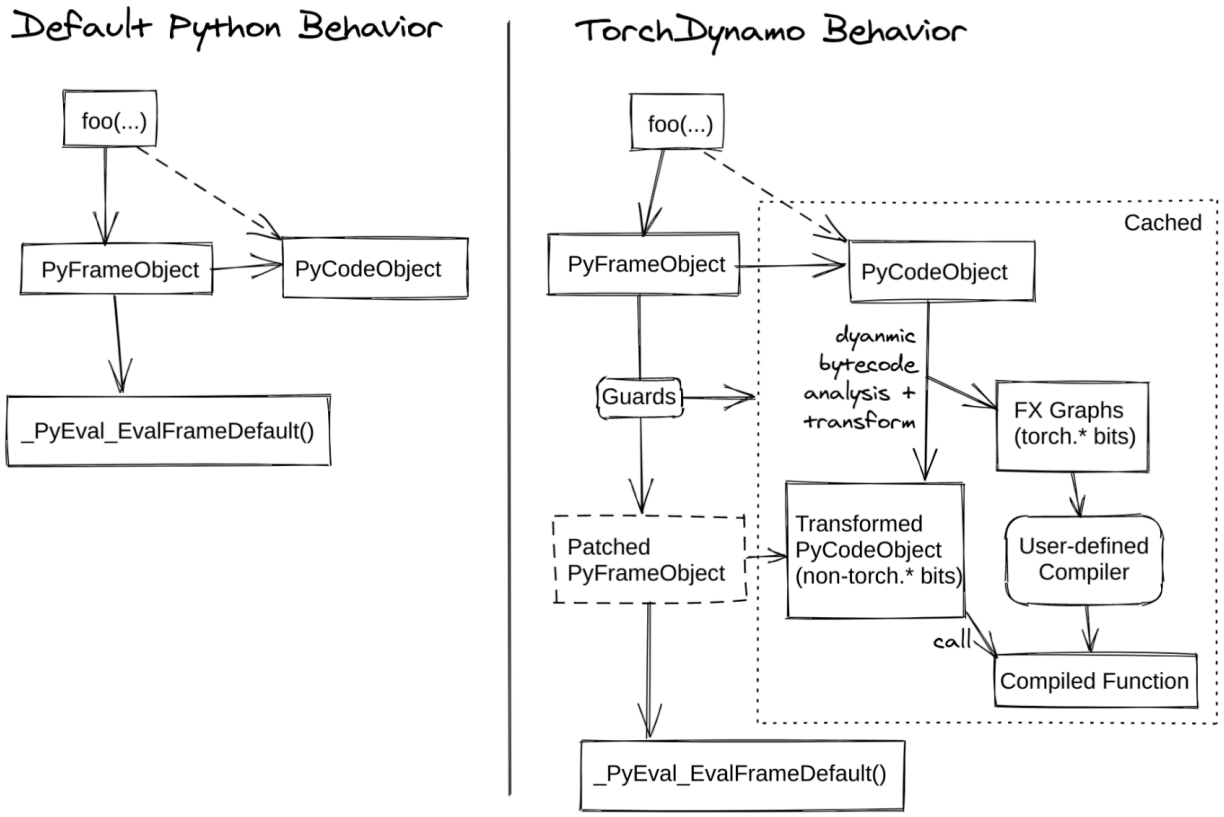
分析 Python 字节码

Dynamo 逐条分析函数的字节码，

- 识别出 PyTorch 操作(torch.* bits)
- 将**连续的 PyTorch 操作**以字节流形式捕获下来
- 若不能捕获则发生**图分割**(Graph Break)

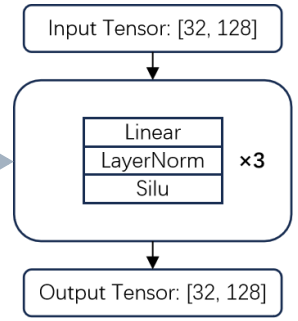
Python 字节码的捕获与转换

拦截和分析 Python 字节码




```python
n, h = 32, 128
repeats = 3
layers = OrderedDict()
for i in range(repeats):
    layers[f"fc_{i}"] = nn.Linear(h, h)
    layers[f"ln_{i}"] = nn.LayerNorm(h)
    layers[f"silu_{i}"] = nn.SiLU()
model = nn.Sequential(layers).cuda().half()

x = torch.randn((n, h), device="cuda", dtype=torch.float16)
compiled = torch.compile(model, mode="reduce-overhead")
with torch.no_grad():
    y = compiled(x)
```



深度学习模型示例

编译前捕获的字节码(compiled 函数)

```
          0 COPY_FREE_VARS       1
 68       2 RESUME               0
 70       4 PUSH_NULL
          6 LOAD_DEREF           2 (fn)
          8 LOAD_FAST            0 (args)
         10 BUILD_MAP            0
         12 LOAD_FAST            1 (kwargs)
         14 DICT_MERGE           1
         16 CALL_FUNCTION_EX     1
         18 RETURN_VALUE
None
```

首次运行时，compiled 内 PyTorch操作都会被捕获，得到字节流形式如下。其中fn即为model

张昱：《编译原理和技术(H)》AI Compiler

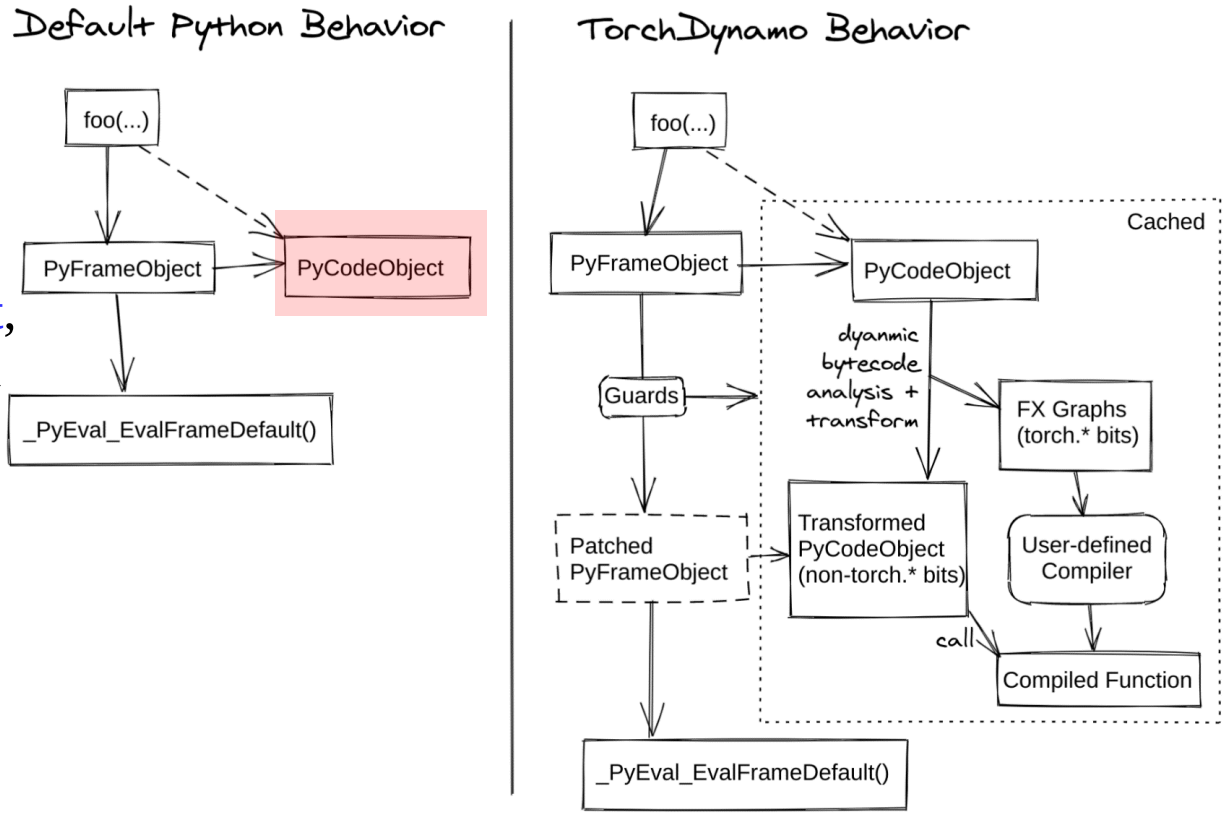
Dynamo 执行的编译后的字节码

- 编译与优化
- 执行编译生成的高性能程序
  - **代码替换**

    Dynamo 创建一个转换后的**PyCodeObject**，替换掉原来的 PyTorch 计算部分，调用编译生成的高效实现

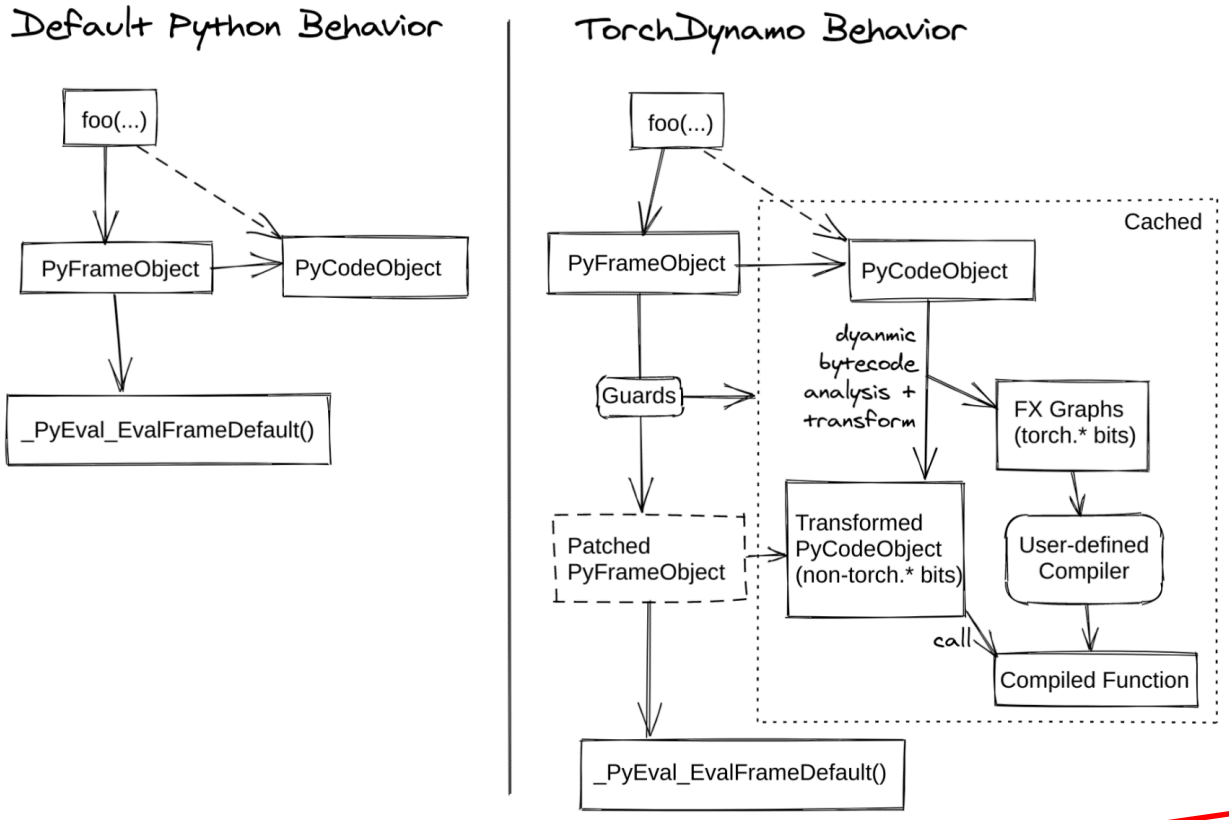
  - **缓存与复用**

    生成的函数都会被缓存。下一次调用 `compiled`，先检查之前的 Guards，若条件都满足（例如输入张量的形状没变），直接运行编译好的函数



Dynamo 执行的编译后的字节码

执行编译生成的高性能程序



编译前捕获的字节码(compiled 函数)

```
          0 COPY_FREE_VARS       1
 68       2 RESUME               0
 70       4 PUSH_NULL
          6 LOAD_DEREF           2 (fn)
          8 LOAD_FAST            0 (args)
         10 BUILD_MAP            0
         12 LOAD_FAST            1 (kwargs)
         14 DICT_MERGE           1
         16 CALL_FUNCTION_EX     1
         18 RETURN_VALUE
None
```

编译完成后得到的字节码(compiled 函数)

```
          # 1. 建立上下文
 68       0 COPY_FREE_VARS       1
          2 RESUME               0
          4 LOAD_GLOBAL         11 (NULL + __compiled_fn_7)
         14 LOAD_GLOBAL         13 (NULL +
             __import_torch_dot__dynamo_dot_utils)
          # 2. 逐层抓取模块参数
          # 3. 调用编译后得到的内核
        610 STORE_FAST          38 (tmp_36)
        612 CALL                13
        620 UNPACK_SEQUENCE      1
        624 RETURN_VALUE
None
```

将 compiled 函数内对fn的调用替换成调用 __compiled_fn_xxx

生成 Torch IR (FX Graph)

分析 Python 字节码 (Bytecode) 并进行符号执行 (Symbolic Execution)，捕获程序中的 PyTorch 操作

将字节码转换为 Fx Graph (Torch IR) 的算法伪代码

```
class InstructionTranslatorBase(
    metaclass=BytecodeDistpatchTableMeta,):
     def step(self):
         ...
         # 符号执行一条字节码指令
         self.dispatch_table[inst.opcode](self, inst)
         ...
```

由字节码符号执行得到的 Fx Graph (Torch IR)

```
class GraphModule(torch.nn.Module):
    def forward(self, ... ):
        ...
        input_1 = torch._C._nn.linear(l_args_0_, ...)
        input_2 = torch.nn.functional.layer_norm(input_1, (128,), ...)
        input_3 = torch.nn.functional.silu(input_2, inplace = False)

        # 剩下的两层网络
        ...
        return (input_9,)
```

输入的model对应的 **Torch IR**

- 动态地构建 FX Graph。这个模拟执行的过程类似于**符号执行**，并未执行模型中的计算操作，避免了实际执行带来的时空开销，同时在模拟过程中完成**常量折叠**和**控制流简化**
- 当遇到动态控制流、函数返回、其他单个计算图中不支持的操作时，发生 “graph break”，Dynamo 会将当前计算图进行切分，先完成当前子图的所有编译阶段，并运行编译结果，之后继续编译后续子图
- Dynamo 的输出是一个 **FX Graph**，这是一种用 Python 代码本身来表示计算图的 IR，我们称之为 **Torch IR**。

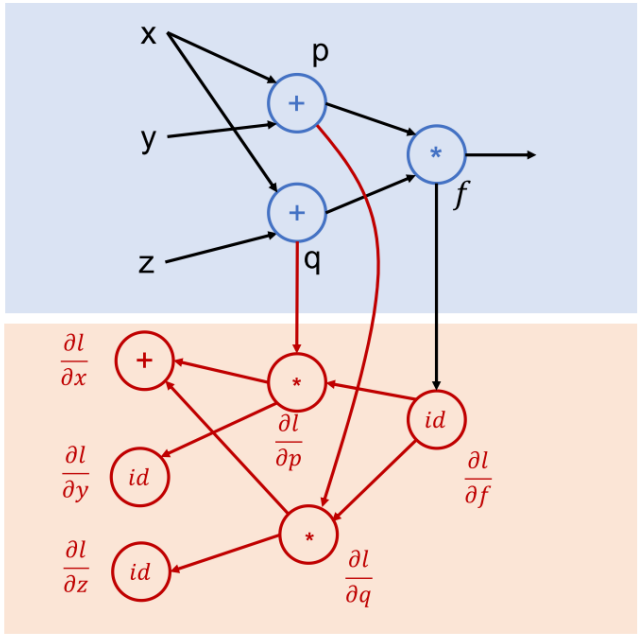
中国科学技术大学
University of Science and Technology of China

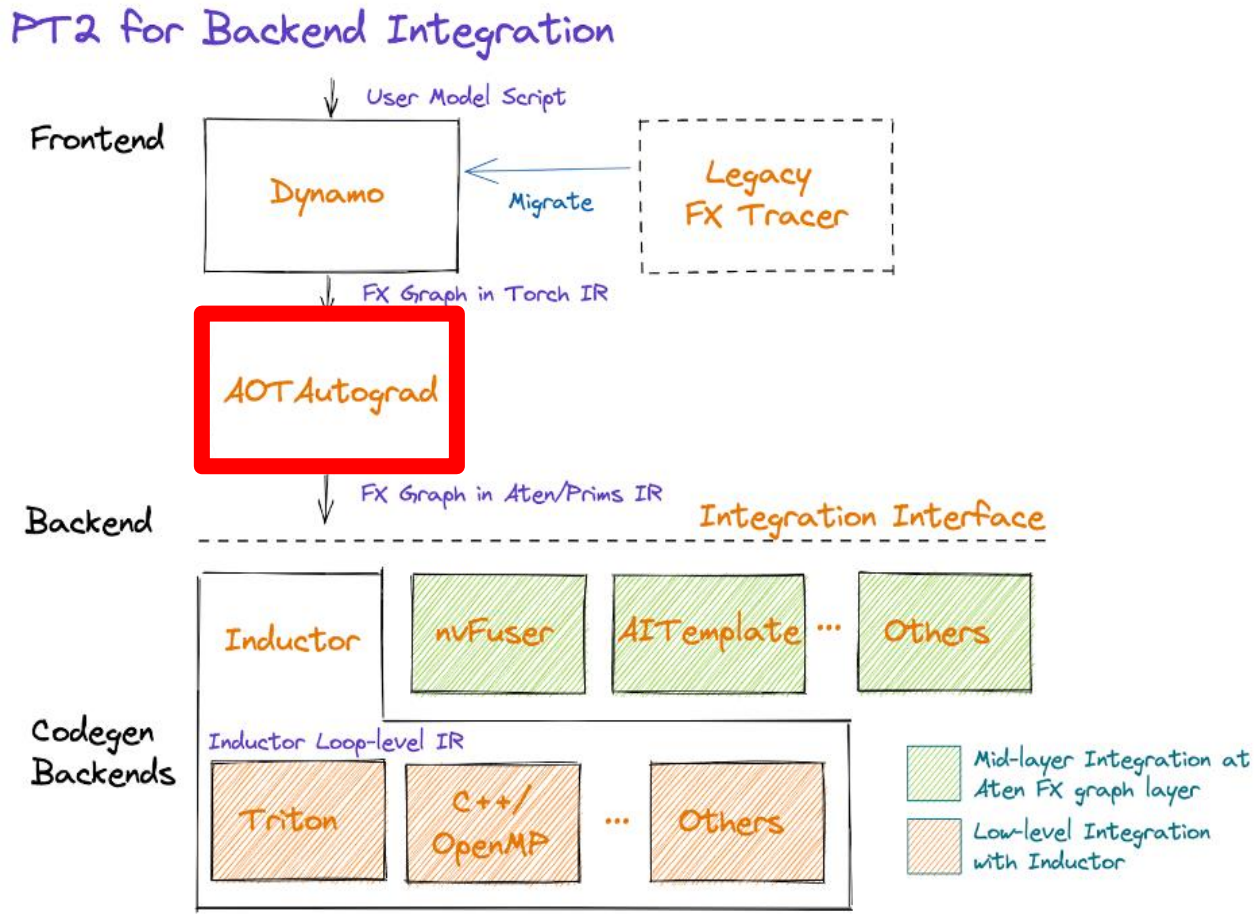
X.1.2 AOT Autograd

第二层：AOT Autograd

针对反向传播的计算图

- 自动微分
  - 不使用`with torch.no_grad()`





- 算子分解

ATen/Prims IR

- 本质上是在 fx.Graph 结构中使用的“算子集”
  - 问题：模型和算法的演进导致定制的算子 API 越来越多，如何统一表示？

    方案：定义元算子集合，用元算子的组合表示一个复杂算子
  - ATen IR
    - Node 的 target 形如 torch.ops.aten.*
    - 对接 PyTorch C++ 后台的核心算子库 ATen 或 Prims IR
  - Prims IR 一个最小化的、规范化的算子集
    - Node 的 target 形如 orch.ops.prims.*
    - Inductor (Triton 后端) 和其他编译器的主要目标 IR
    - 可以由 ATen IR 分解而来,节点代表的是最基本的计算单元

      如 add, mul, rsqrt, reduce, view, broadcast

输出日志

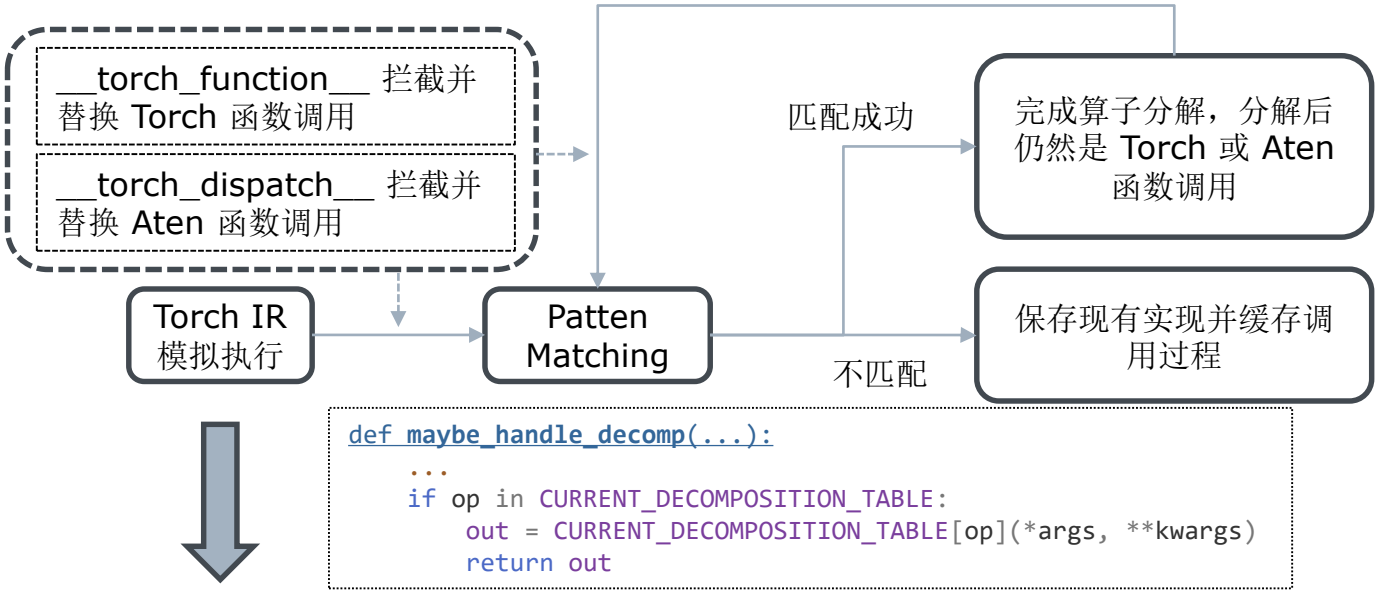
```
class <lambda>(torch.nn.Module):
    def forward(self, ...):
        ...
        permute: "f16[128, 128]" =
             torch.ops.aten.permute.default(arg0_1, [1, 0])
        mm_default_2: "f16[32, 128]" =
             torch.ops.aten.mm.default(arg2_1, permute)
        add_tensor_2: "f16[32, 128]" =
             torch.ops.aten.add.Tensor(mm_default_2, arg1_1)
        ...
        return (convert_element_type_20,)
```

FX Graph 下降到 ATen IR 的 Python 代码示例

FX Graph 下降到 ATen/Prims IR

对自动微分后的 FX Graph，执行算子分解

- 算子分解: 将 Torch IR 中的复合 PyTorch 算子（例如 nn.Linear）分解为更基础、更原子的算子集合。这个标准的算子集被称为 ATen/Prims IR




```
# 将 python api 分解为 ATen 级算子后的 Fx Graph(Aten/Prims IR)
class <lambda>(torch.nn.Module):
    def forward(self, arg0_1: "f16[128, 128]", arg1_1: "f16[128]", arg2_1: "f16[32, 128]", arg3_1: "f16[128]", arg4_1: "f16[128]", arg5_1: "f16[128, 128]",
arg6_1: "f16[128]", arg7_1: "f16[128]", arg8_1: "f16[128]", arg9_1: "f16[128, 128]", arg10_1: "f16[128]", arg11_1: "f16[128]", arg12_1: "f16[128]"):
        permute: "f16[128, 128]" = torch.ops.aten.permute.default(arg0_1, [1, 0])
        mm_default_2: "f16[32, 128]" = torch.ops.aten.mm.default(arg2_1, permute)
        add_tensor_2: "f16[32, 128]" = torch.ops.aten.add.Tensor(mm_default_2, arg1_1)
        ...
        return (convert_element_type_20,)
```

对应torch IR中的一个 torch._C._nn.linear

中国科学技术大学

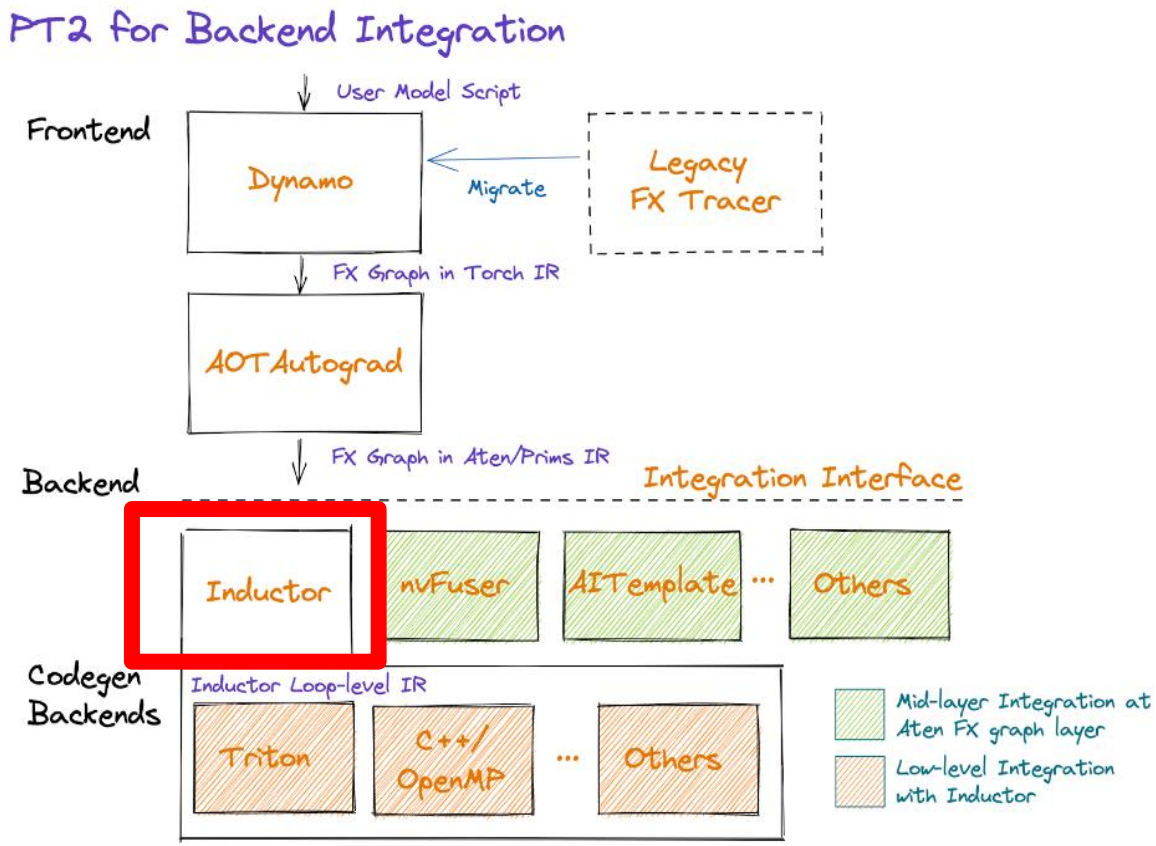
University of Science and Technology of China

X.1.3 Inductor

第三层：Inductor

Inductor 将 ATen/Prims IR 进一步降低到 Loop-level IR，并最终生成目标硬件上的特点程序

- **核心：生成 Loop-level IR**
  - 描述算子内的循环嵌套
  - 在 2025.11，ATen/Prims 中有 315 个算子，需要一套程序语言来描述这些算子的逻辑
- **基于 Loop-Level IR 生成后端代码**
  - 为 GPU 生成 Triton
  - 为 CPU 生成 C++/OpenMP



Loop-level IR

- **描述循环的 IR：Loop-level IR**
  - Prims IR 中一组融合节点被包装为一个 ComputedBuffer
    - 维护数据流依赖、数据类型和数据形状、计算的定义
    - 计算的定义：内部的每个 Prims IR 节点用一个 Loop 表示
  - 计算的定义 **“输出缓冲区中每个元素是如何计算的”**
    - 一般情况下，通过 PointWise, Reduction, Scan, Sort 描述各种类型的循环
    - 复杂情况：不规则、稀疏访存等；通过 ExternKernel 直接调用 ATen 算子库，或 TritonTemplateKernel 直接集成 Triton 代码
  - 最终程序由一系列调度节点 **SchedulerNode** 组成，这些节点是进行优化和代码生成的基本单元
    - **SchedulerNode** 包含 ComputedBuffer, ExternKernel, TritonTemplateKernel, InputBuffer (输入)

```
Node(
  name="add_1",
  op="call_function",
  target=OpOverload(op='aten.add', overload='Tensor'),
  args=(c, 1),
  users={output: None},
)
```

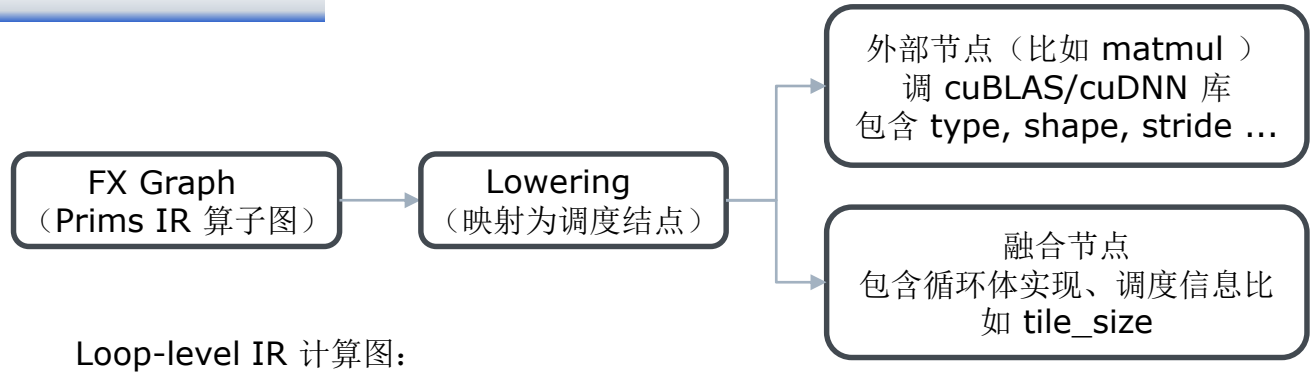
d=c+1 对应的 prims IR

```
ComputedBuffer (
  'cuda:0',        # 数据所在的设备
  torch.float32,   # 数据类型
  def inner_fn(index):          # 计算逻辑
    i0, i1 = index              # 索引变量
    tmp0 = ops.load(buf0, i1 + 512 * i0)
    tmp1 = ops.constant(1, torch.float32)
    tmp2 = tmp0 + tmp1
    return tmp2
  ,
  ranges=[32, 512],             # 索引变量的范围
  ...
)
```

d=c+1 对应的 Loop-Level IR

Graph 下降到 Loop-level IR

- **Loop-level IR 是优化的关键**
  - 算子融合，例如：将多个逐点（Pointwise）算子融合成一个核函数，从而减少内存读写开销
  - 其他优化
    - 内存布局优化
    - 循环分块与重排
    - 死代码删除
    - 缓冲区管理与复用
    - 并行
    - ...



Loop-level IR 计算图：

```
[..., SchedulerNode(nodes=op1_op2_op4_op5), ...]
```

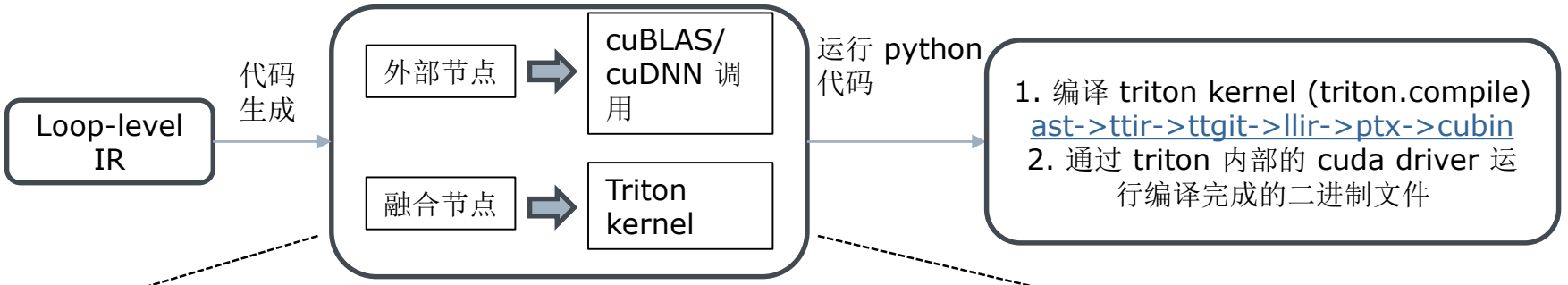
输出日志

```
var_ranges = {p0: 32, p1: 128}  # shape dimensions
index0 = 128*p0 + p1
index1 = p1
index2 = p0
def body(self, ops):  # 算子的循环体
    get_index = self.get_index('index0')
    load = ops.load('buf0', get_index)
    get_index_1 = self.get_index('index1')
    load_1 = ops.load('arg1_1', get_index_1)
    add = ops.add(load, load_1)
    to_dtype = ops.to_dtype(add, torch.float32, src_dtype = torch.float16)
    reduction = ops.reduction(torch.float32, torch.float32, 'welford_reduce', to_dtype)
    getitem = reduction[0]
    getitem_1 = reduction[1]
    getitem_2 = reduction[2]
    get_index_2 = self.get_index('index2')
    store_reduction = ops.store_reduction('buf1', get_index_2, getitem)
    return store_reduction
```

Loop-level IR 计算图的某个融合算子的op1，执行 layernorm 中的 reduction_add 加法

代码生成 (Triton / C++)

Loop-level IR → Triton/C++ Kernels



编译 triton kernel

生成的 triton kernel
addmm_native_layer_norm_silu

生成的 kernel code

```
triton_per_fused_addmm_native_layer_norm_silu_0 =
async_compile.triton('triton_per_fused_addmm_native_layer_norm_
silu_0', '''
import triton
import triton.language as tl
@triton.jit
def triton_per_fused_addmm_native_layer_norm_silu_0(in_ptr0,
in_ptr1, in_ptr2, in_ptr3, out_ptr3, xnumel, r0_numel, XBLOCK :
tl.constexpr):
    xnumel = 32
    r0_numel = 128
    R0_BLOCK: tl.constexpr = 128
    rnumel = r0_numel
    RBLOCK: tl.constexpr = R0_BLOCK
    xoffset = tl.program_id(0) * XBLOCK
    ...
    tl.store(out_ptr3 + (r0_1 + 128*x0), tmp35, xmask)
''', device_str='cuda')
```

生成的 wrapper code

```
...
buf0 = empty_strided_cuda((32, 128), (128, 1),
torch.float16)
extern_kernels.mm(arg2_1,
reinterpret_tensor(arg0_1, (128, 128), (1,
128), 0), out=buf0)
buf5 = empty_strided_cuda((32, 128), (128, 1),
torch.float16)
stream0 = get_raw_stream(0)
triton_per_fused_addmm_native_layer_norm_silu_
0.run(buf0, arg1_1, arg3_1, arg4_1, buf5, 32,
128, stream=stream0)
...
```

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

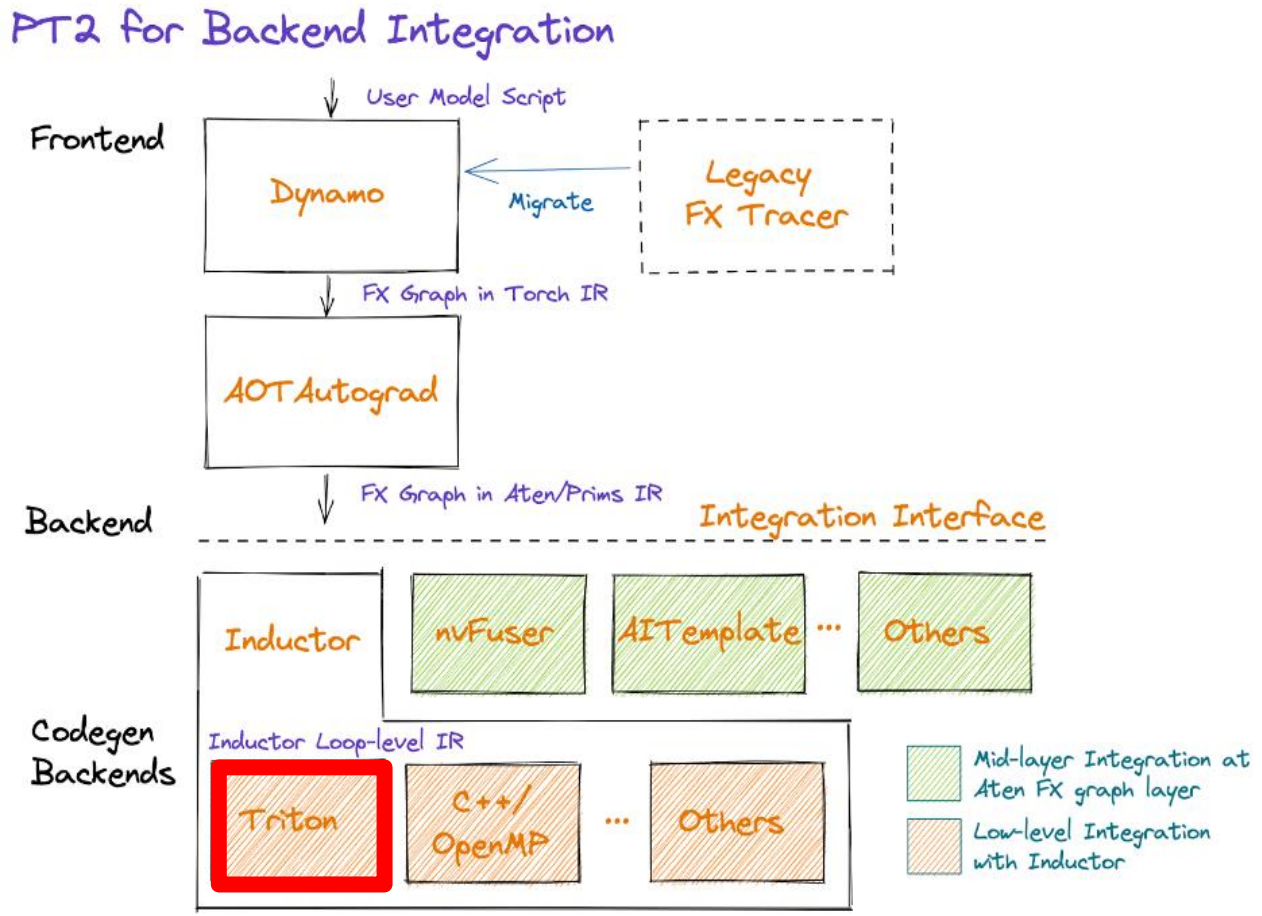
X.2 算子编译器：Triton

- **Triton**[MAPL 2019]

什么是Triton

A programming language for highly performant GPU kernels

- **层级高于 CUDA，低于已有的张量程序DSL**
- **便于用户实现高效自定义算子**
  - 相较于Torch DSL，Triton 允许用户在 SRAM 中定义张量（数据分块），并且可以用 Torch-like API 修改这些分块
  - Triton 编译器完成后续的优化



[MAPL 2019] Triton: an intermediate language and compiler for tiled neural network computations
https://openai.com/index/triton/

Triton 程序示例

Triton DSL 特点

- 并发：SPMD模型，简化并行编程
  - `tl.program_id`
- 内存：编译器自动管理缓存，牺牲手动控制换取开发效率
  - `tl.load`, `tl.store`
- 编译：constexpr 实现编译期优化，以牺牲运行时灵活性换取高性能
  - `tl.constexpr`

```
def triton_per_fused_addmm_native_layer_norm_silu_0(in_ptr0,
        in_ptr1,in_ptr2,in_ptr3,out_ptr3,XBLOCK:tl.constexpr):
    xnumel=32
    R0_BLOCK:tl.constexpr=128
    xoffset=tl.program_id(0)*XBLOCK
    xindex=xoffset+tl.arange(0,XBLOCK)[:,None]
    xmask=xindex<xnumel
    r0_index=tl.arange(0,R0_BLOCK)[None,:]
    r0_1=r0_index
    x0=xindex
    tmp0=tl.load(in_ptr0+(r0_1+128*x0),xmask,other=0.0)
    tmp1=tl.load(in_ptr1+(r0_1),None)
    tmp27=tl.load(in_ptr2+(r0_1),None)
    tmp30=tl.load(in_ptr3+(r0_1),None)
    tmp2=tmp0+tmp1
    tmp3=tmp2
    tmp4=tl.broadcast_to(tmp3,[XBLOCK,R0_BLOCK])
    tmp7=tl.broadcast_to(tmp4,[XBLOCK,R0_BLOCK])
    tmp9=tl.where(xmask,tmp7,0)
    tmp10=tl.sum(tmp9,1)[:,None]
    ... # 为简洁，省略后续计算逻辑
    tl.store(...)
```

tmp10代表layernorm计算过程中的第一步，代表了输入张量在特征维度（最后一维）上的元素之和。

Triton IR

Triton IR (TT IR)是 tile-level 的 MLIR 方言

- 依托 MLIR 基础设施
  - SSA 形式（静态单赋值）
  - 基本块与控制流图 CFG
  - 类型系统
  - Dialect 机制
- Tile-level 的 IR
  - Tile 是一个 小块张量（例如 16×32、8×128）

    Device 中一个计算核心处理的局部数据分块
  - 运算是 tile 粒度，而非 scalar 粒度

    例如：控制流通过 mask 实现，避免分支指令
- 依托 Triton IR 非常方便开展 Tile-level 的优化
- 执行硬件无关优化的载体

```
%a = tt.load %ptr_a : memdesc<16x32xf16>
%b = tt.load %ptr_b : memdesc<16x32xf16>
%c = arith.mulf %a, %b : tensor<16x32xf16>
```

Triton IR 操作的数据对象是 data Tile

```
%mask = tt.compare …
%val  = tt.load %ptr, %mask
```

包括控制流在内的运算是 Tile 粒度的

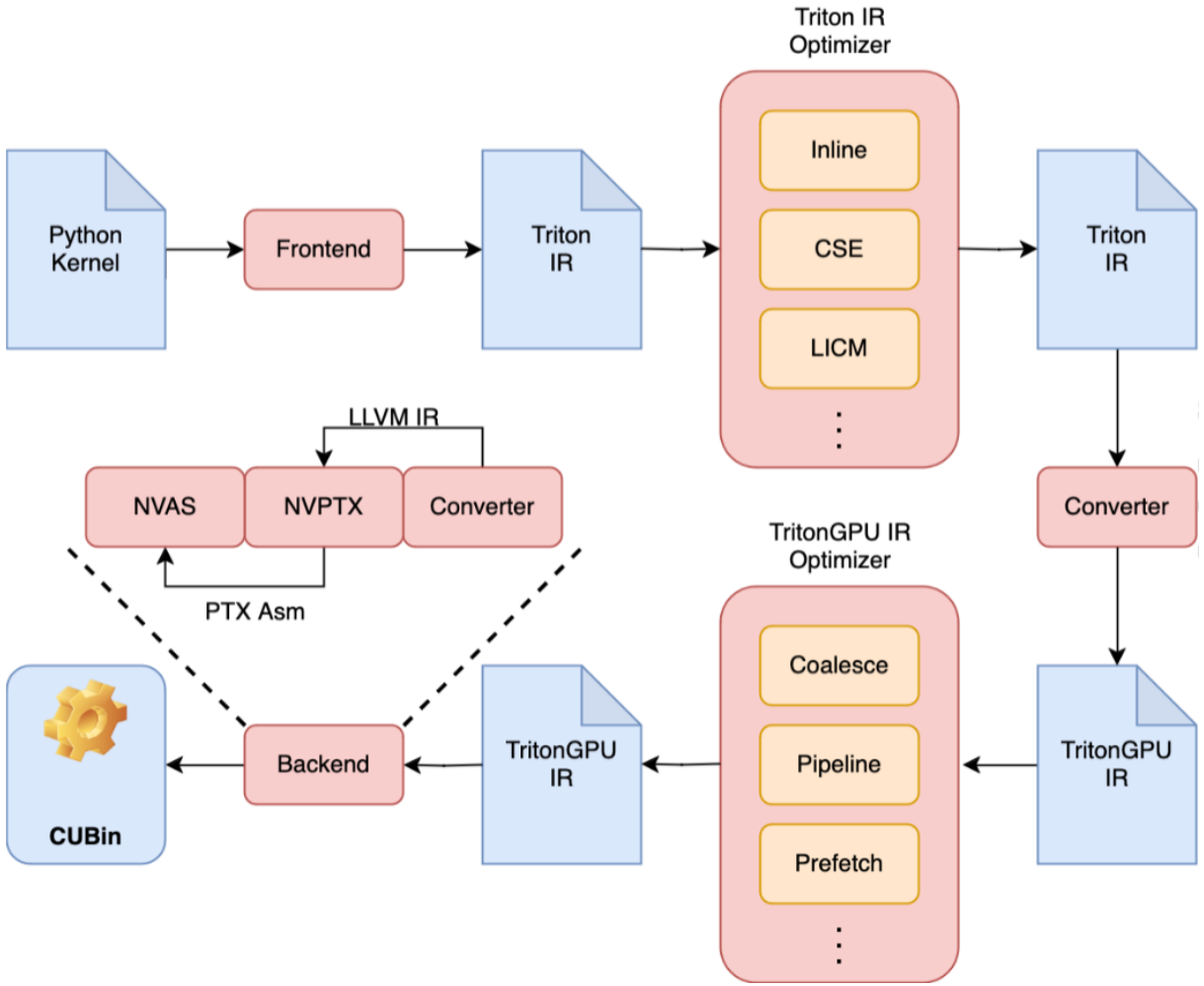
Tile 抽象驱动的自动优化

- 将离散访存转为连续访存
- 利用 tile-level 属性推导分块/并行策略
- 共享内存调度（比如多个 tile 复用同一个 SRAM、先重排再加载进 SRAM）

Triton 编译器

Triton 编译流程：

- 前端：
  - 访问 Python kernel code 的抽象语法树（AST），转换为 Triton IR
- 代码优化：
  - 通过各类 pass 将 Triton IR 逐步转换为优化过的 TritonGPU IR
    - 通用优化：Passes
    - 硬件无关优化：Passes
    - GPU特定优化：Passes
- 后端：
  - 将 TritonGPU IR 逐步转换为 LLVM IR，并最终编译为 cubin



Triton 代码优化

- **公共优化（LLVM/MLIR的优化Pass）**
  - 常量传播，死代码删除，函数内联，规范化，公共子表达式删除，循环不变量外提，...
- **Triton IR Optimizer**
  - Combine：针对某些特定 pattern 的重写优化
    - 比如把 sum(x[:, :, None] * y[None, :, :], 1) 重写为 dot(x, y)
  - 循环展开, Reorder, 循环不变代码的外提,...
- **TritonGPU IR Optimizer**
  - 例如：Pipeline（global to shared memory 的多缓冲区优化）, Prefetch（shared memory to 寄存器的多缓冲区优化），...

[MAPL 2019] Triton: an intermediate language and compiler for tiled neural network computations
https://openai.com/index/triton/

Triton GPU IR 特点

- Triton GPU IR (TTG IR)是 block-level 的 IR，用于表达 GPU kernel 的行为
- 相较于 Triton IR，引入：
  - 硬件相关的数据布局属性：

    描述线程、warp、CTA等硬件抽象的数据布局以及维度顺序

  - 数据布局转换

    例如 ConvertLayout 为数据 Tile 指定内存，如指定使用 shared memory 或寄存器

  - 内存与指针操作
  - 硬件特定算符

    例如 tc_gen5_mma 封装了 Tensor Core 的 mma.sync 或 wgmma 指令的操作

```
// 一个 1024x1024 的 f16 块，在 TTGIR 中表示为：
tensor<1024x1024xf16, #triton_gpu.blocked<{
   sizePerThread = [4, 8],
   threadsPerWarp = [4, 8],
   warpsPerCTA = [4, 1],
   order = [1, 0]
}>>
```

- 执行硬件相关优化的载体
  - 不同硬件通常会提出不同的方言，来扩充硬件相关优化的能力，例如
    - Nvidia TritonNvidiaGPUOps, AMD TritonAMDGPUOps

各种IR的特点

IR	层级	特点	动态形状	用途
FX Graph IR	Python-level graph	表达高层语义	支持 内置符号变量和相关运算符，维护符号表达式	捕获模型图结构
Aten IR	Operator-level	语义固定，对应 PyTorch ATen 算子库	支持	表示较高层级的算子
Prims IR	Math primitive	更加规范化的元算子集合	支持	表示较低层级的算子
Loop-level IR	Loop-level	显式表达循环、索引、load/store、计算操作	支持 符号被视作变量	kernel 融合等图级别优化
Triton IR	MILR Dialect	高层级张量程序抽象，block/grid 索引，自动优化	支持 与变量相关的控制流被转换为 mask 或 conditional 算符	表达硬件无关的张量程序逻辑
Triton GPU IR	LLVM IR	包含硬件指令级别。含有硬件特定优化，如寄存器有优化、shared/global memory, ...	支持 mask 运算或 predictive 指令	表达硬件相关的张量程序逻辑

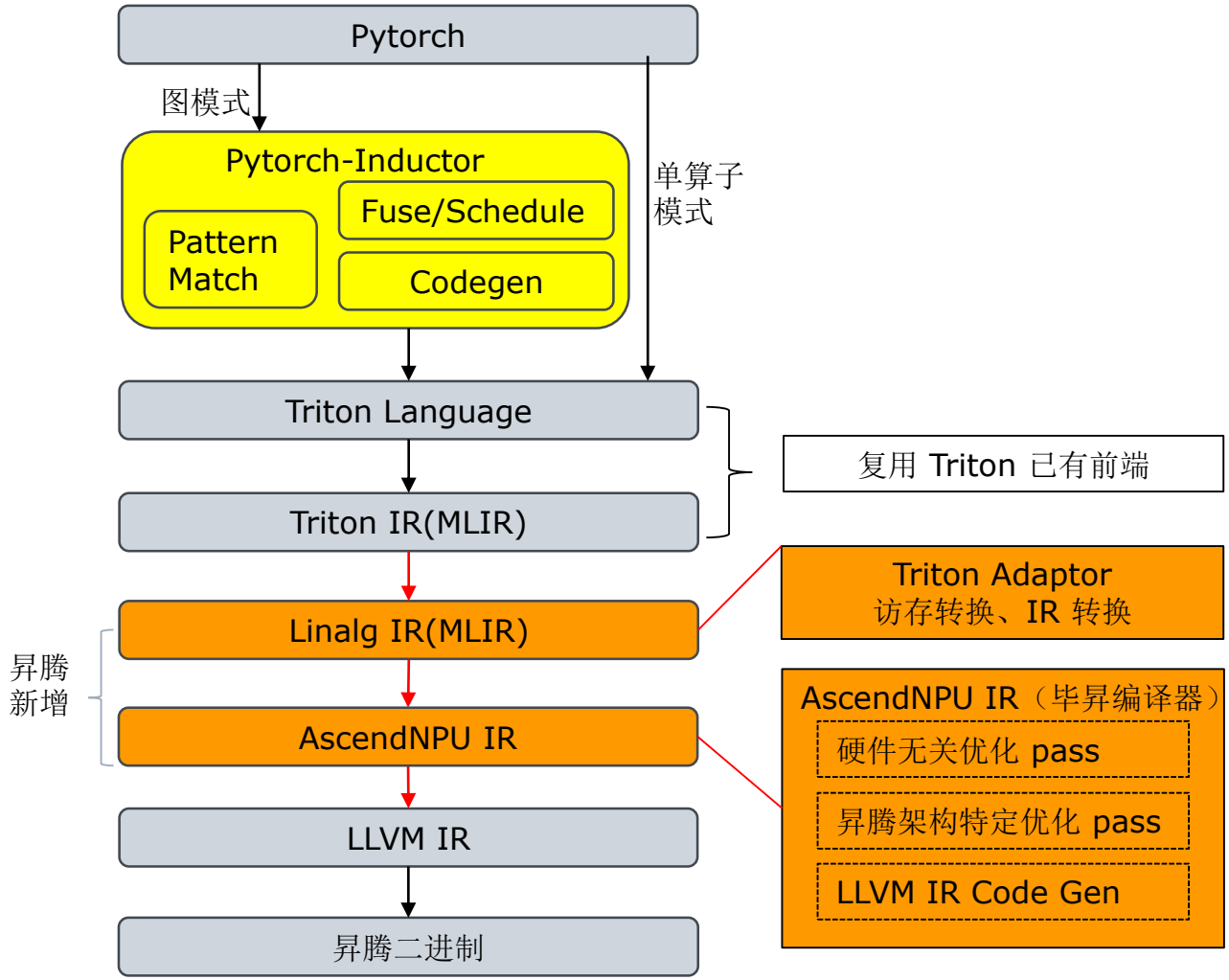
中国科学技术大学
University of Science and Technology of China

X.3 昇腾编译介绍

Ascend AI 编译 以Triton Ascend 为例

Triton-Ascend

- 支持单算子模式编译
  - 复用 Triton 已有前端，将 Python AST 转为 Triton IR
  - Triton Adaptor 将离散访存转为连续线性访存，并将 Triton IR 转成 Linalg IR
  - 毕昇编译器通过各类 Pass 将 Linalg IR 转成优化后的 AscendNPU IR，并生成 LLVM IR 代码，最终编译得到 kernel.o
- Torch-NPU Inductor 对接 Triton-Ascend
  - 最新: 2.6.0 已支持



Ascend Bisheng 编译器

- **CANN 内置异构编程器**
  - 基于 LLVM 15.0.5 实现

    毕昇编译器介绍 毕昇编译器工具链下载
  - 支持 Ascend C编程语言，内置基础API和高阶API，支持 CATLUSS 模板库
  - 通过 **AscendNPU IR** 接入 Triton 编译流程
  - 在编译结束后，由驱动发射、运行 kernel.o