AI编译器系列

AI 编译器通用架构

ZOMI

Talk Overview

1. 传统编译器
   - History of Compiler - 编译器的发展
   - GCC process and principle – GCC 编译过程和原理
   - LLVM/Clang process and principle – LLVM 架构和原理
2. **AI编译器**
   - History of AI Compiler – 为什么出现AI编译器
   - AI Compiler Three Stage – AI编译器的发展
   - Base Common architecture – AI编译器的通用架构
   - Different and challenge of the future – 未来的挑战与思考

Question?

1. AI 编译器跟传统编译器有什么区别吗？
2. AI 框架跟AI编译器什么关系？是AI框架包含AI编译器 or AI框架就是一个AI编译器？
3. AI 领域真的需要编译器吗？那为什么 PyTorch 动态图模式没有编译器的概念？
4. 技术投入比来看，神经网络编译器和人工算子实现，哪个性价比更高？

现有 AI 编译器架构

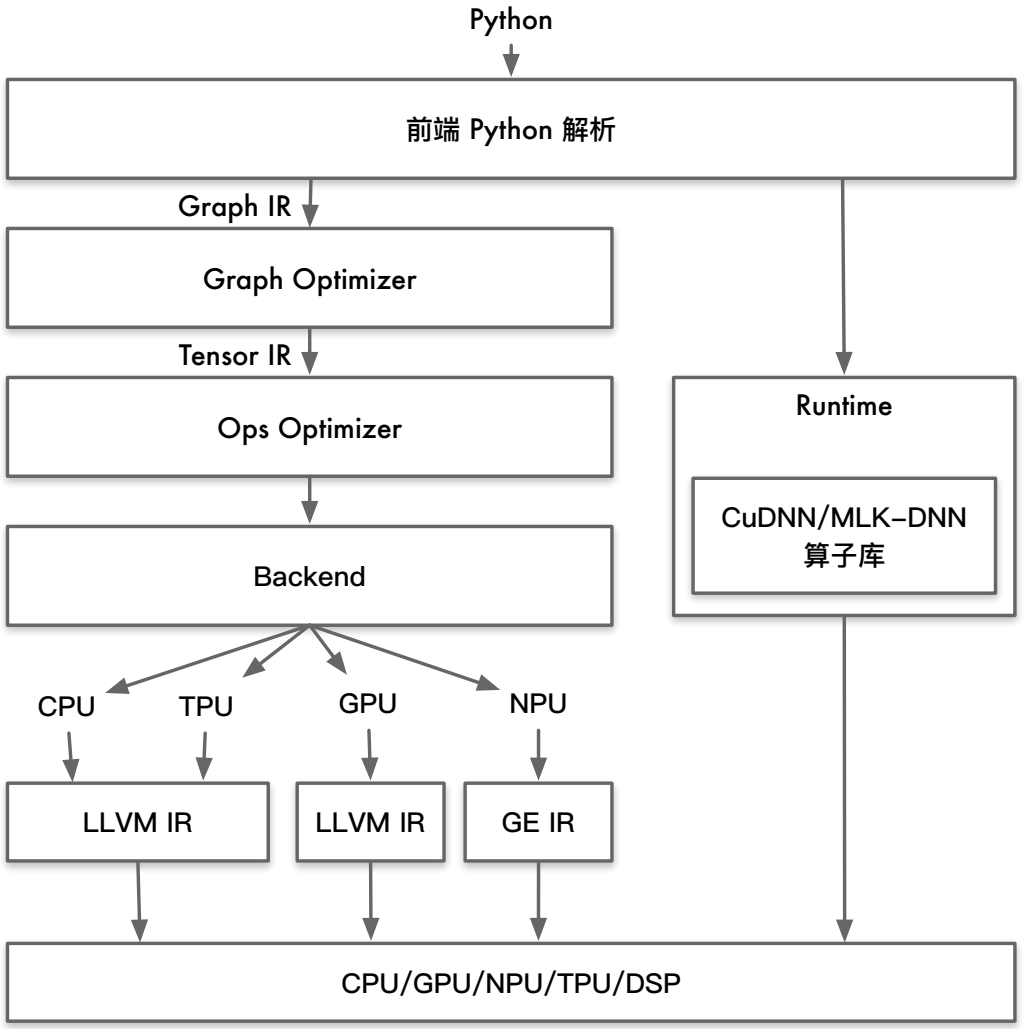
development history: Stage II 专用的AI编译器

表达上：

- 以 PyTorch 为标杆的表达转换到计算图层 IR 进行优化。

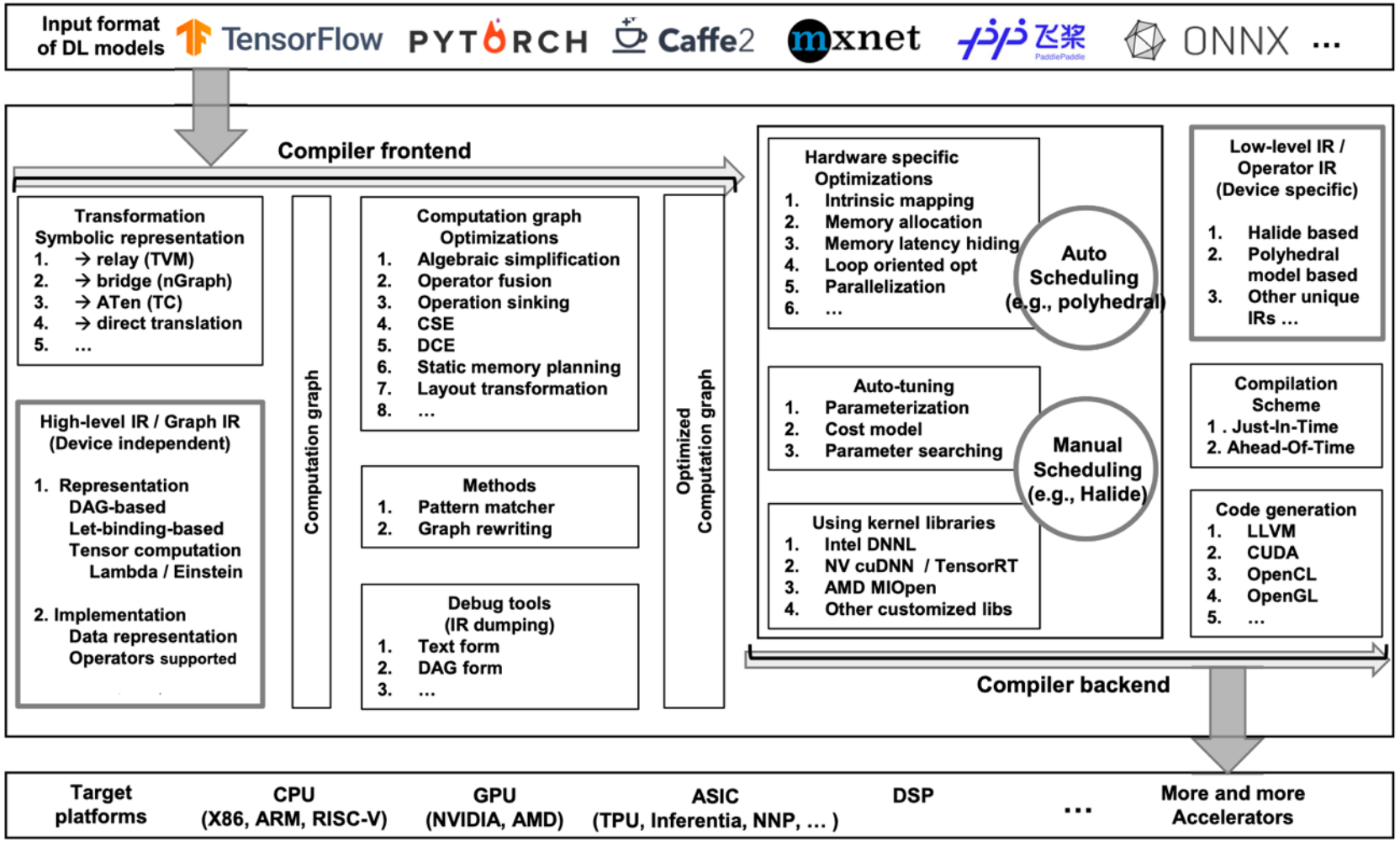
性能上：

- 打开计算图和算子的边界，进行重新组合优化，发挥芯片的算力。



What is AI Compiler?

The Deep Learning Compiler: A Comprehensive Survey



IR 中间表达

编译器主要分为前后端，分别针对于硬件无关和硬件相关的处理。每一个部分都有自己的 IR (Intermediate Representation，中间表达)，每个部分也会对进行优化：

- High-level IR：用于表示计算图，其出现主要是为了解决传统编译器中难以表达深度学习模型中的复杂运算这一问题，为了实现更高效的优化所以新设计了一套 IR。
- Low-level IR：能够在更细粒度的层面上表示模型，从而能够针对于硬件进行优化，文中将其分为了三类。

Frontend 前端优化

构造计算图后，前端将应用图级优化。因为图提供了计算全局概述，所以更容易在图级发现和执行许多优化。前端优化与硬件无关，这意味着可以将计算图优化应用于各种后端目标。前端优化分为三类：

1. 节点级优化，如 Zero-dim-tensor elimination、Nop Elimination
2. 块级优化，如代数简化、常量折叠、算子融合
3. 数据流级优化，如Common sub-expression elimination、DCE

Backend 后端优化

特定硬件的优化

- 目标针对特定硬件体系结构获取高性能代码。1）低级IR转换为LLVM IR，利用LLVM基础结构生成优化的CPU/GPU代码。2）使用领域知识定制优化，这可以更有效地利用目标硬件。

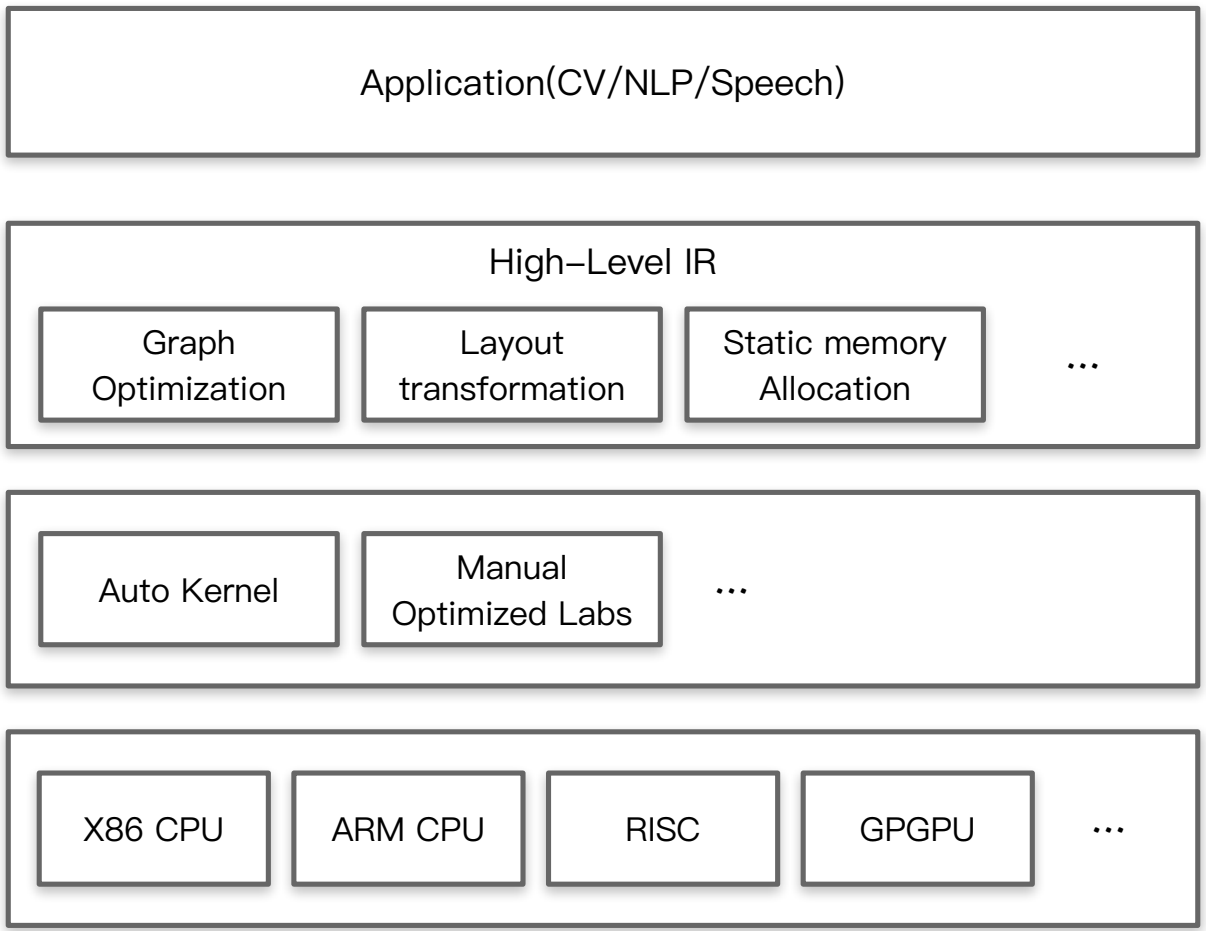
自动调整

- 由于在特定硬件优化中用于参数调整的搜索空间巨大，因此有必要利用自动调整来确定最佳参数设置。1）Halide/TVM允许调度和计算表达分开，使用自动调节来得出较佳配置。2）应用多面体模型 Polyhedral model 进行参数调整。

优化内核库

- 厂商特定优化内核库，广泛用于各种硬件上的加速DL训练和推理。特定优化原语可以满足计算要求时，使用优化的内核库可显著提高性能，否则可能会受到进一步优化的约束。

What is AI Compiler?



DL Models

Graph Level

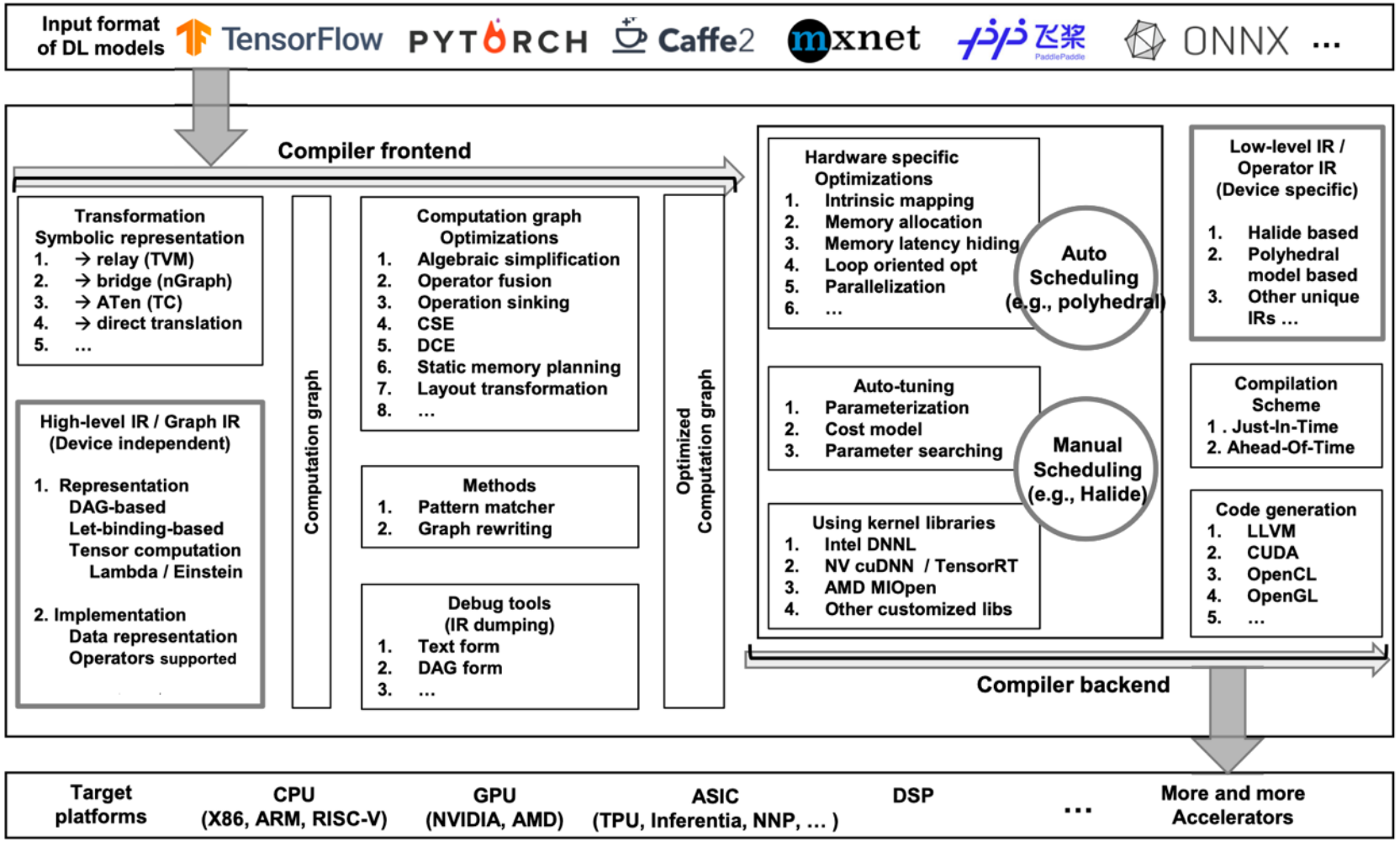
Kernel Level

Hardware

Summary

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

Summary


Input format of DL models
TensorFlow
PYTORCH
Caffe2
mxnet
飞桨
ONNX
...
Compiler frontend
Transformation
Symbolic representation
1. → relay (TVM)
2. → bridge (nGraph)
3. → ATen (TC)
4. → direct translation
5. ...
High-level IR / Graph IR
(Device independent)
1. Representation
DAG-based
Let-binding-based
Tensor computation
Lambda / Einstein
2. Implementation
Data representation
Operators supported
Computation graph
Computation graph
Optimizations
1. Algebraic simplification
2. Operator fusion
3. Operation sinking
4. CSE
5. DCE
6. Static memory planning
7. Layout transformation
8. ...
Methods
1. Pattern matcher
2. Graph rewriting
Debug tools
(IR dumping)
1. Text form
2. DAG form
3. ...
Optimized Computation graph
Hardware specific Optimizations
1. Intrinsic mapping
2. Memory allocation
3. Memory latency hiding
4. Loop oriented opt
5. Parallelization
6. ...
Auto Scheduling (e.g., polyhedral)
Auto-tuning
1. Parameterization
2. Cost model
3. Parameter searching
Manual Scheduling (e.g., Halide)
Using kernel libraries
1. Intel DNNL
2. NV cuDNN / TensorRT
3. AMD MIOpen
4. Other customized libs
Low-level IR / Operator IR
(Device specific)
1. Halide based
2. Polyhedral model based
3. Other unique IRs ...
Compilation Scheme
1 . Just-In-Time
2. Ahead-Of-Time
Code generation
1. LLVM
2. CUDA
3. OpenCL
4. OpenGL
5. ...
Compiler backend
Target platforms
CPU (X86, ARM, RISC-V)
GPU (NVIDIA, AMD)
ASIC (TPU, Inferentia, NNP, ...)
DSP
...
More and more Accelerators