AI编译器系列

AI 编译器架构发展

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 – 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

什么是 AI 编译器

什么是 AI 编译器

- **推理场景：**输入 AI 框架训练出来的模型文件，输出能够在不同硬件高效执行的程序；
- **训练场景：**输入高级语言表示的神经网络代码，输出能够在不同硬件高效执行的程序；

Question?

- 什么是训练场景？什么是推理场景吗？
- 搞 AI 编译器为什么要了解算法呢？
- 搞 AI 算子为什么要了解编译器？

What is AI Compiler?

1. Python 为主的动态解释器语言前端
2. 多层 IR 设计，包括图编译、算子编译、代码生成
3. 面向神经网络、深度学习的特定优化
4. DSA 芯片架构的支持

development history

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

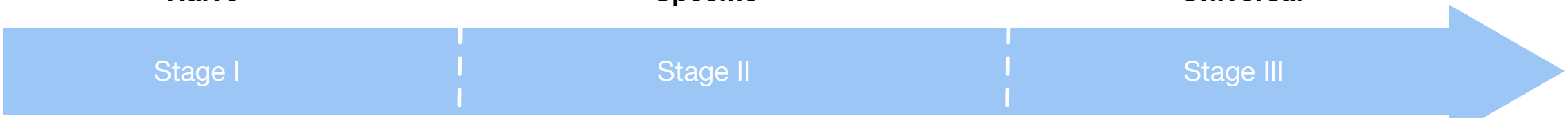
Naive

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

Specific

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

Universal



Caffe

Cons

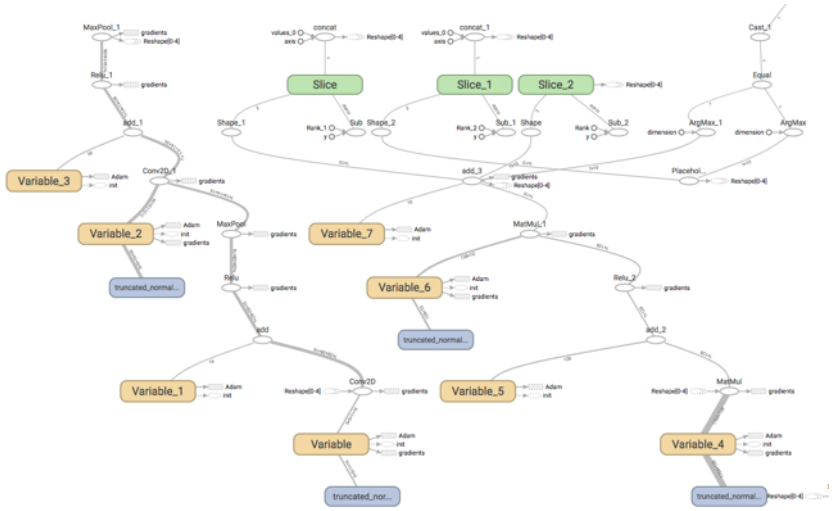
1. API构图，易用性差
2. 大量DSA异构芯片对性能挑战
3. 算子边界确定无法充分发挥性能

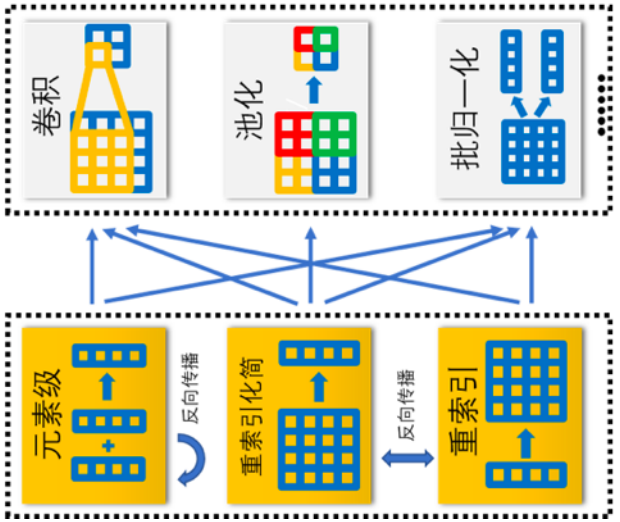
Cons

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

development history: Stage I 朴素的AI编译器

- TensorFlow 早期版本，基于神经网络的编程模型，主要进行了graph 图和ops 算子两层抽象。
  - **图层：**通过声明式的编程方式，以静态图方式执行，执行前进行硬件无关和硬件相关的编译优化。硬件无关的优化，如表达式化简、常量折叠、自动微分等；硬件相关的优化包括算子融合、内存分配等。
  - **算子层：**通常采用手写 kernel 的方式，如在 NVIDIA GPU 上基于 CUDA kernel 实现大量的 .cu 算子或者依赖于 CuDNN 算子优化库。





development history: Stage I 朴素的AI编译器

- TensorFlow 早期版本，基于神经网络的编程模型，主要进行了graph 图和ops 算子两层抽象。
  - **图层**：通过声明式的编程方式，以静态图方式执行，执行前进行硬件无关和硬件相关的编译优化。硬件无关的优化，如表达式化简、常量折叠、自动微分等；硬件相关的优化包括算子融合、内存分配等。
  - **算子层**：通常采用手写 kernel 的方式，如在 NVIDIA GPU 上基于 CUDA kernel 实现大量的 .cu 算子或者依赖于 CuDNN 算子优化库。

【AI框架基础】系列第四篇！函数式编程和声明式编程有...

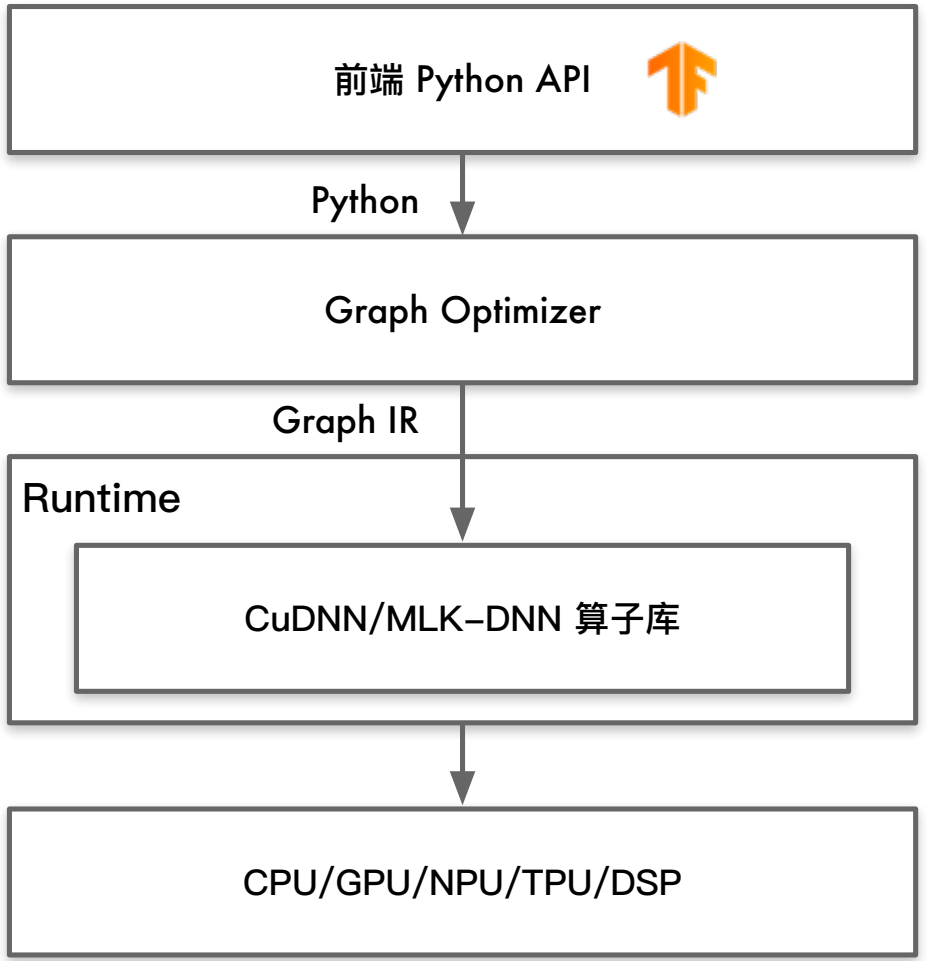
22-10-06 01:53:44 当前字幕: 1（含平台自动生成）

135 1 2 2 5 11 0

https://www.bilibili.com/video/BV1gR4y1o7WT/

development history: Stage I 朴素的AI编译器

- TensorFlow 早期版本，基于神经网络的编程模型，主要进行了graph 图和ops 算子两层抽象；



development history: Stage I 朴素的AI编译器

表达上：

- 静态图的表达式非 Python 原生，开发者主要通过框架提供Python API 显示构图，易用性上不好；

性能上：

- DSA 专用加速芯片出现加剧了性能上的挑战；
- 算子层提供的算子粒度和边界提前确定后，无法充分发挥硬件的性能；
- 硬件厂商的提供的算子优化库也未必最优
  - 1）模型和 shape 确定情况下，可能还有更优算子实现；
  - 2）在 SIMT 和 SIMD 架构中，Scheduling、Tilling 都有有很大的空间。

https://github.com/pytorch/pytorch/tree/master/aten

development history: Stage II 专用的AI编译器

表达上：

- PyTorch 灵活表达 API 方式成为 AI 框架参考标杆，图层的神经网络编译器主要就是考虑如何把类 PyTorch 的表达转换到图层的IR进行优化。
- 类PyTorch的Python原生表达，静态化转换；
- AI专用编译器架构，打开图算边界进行融合优化；

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))
```

development history: Stage II 专用的AI编译器

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。计算图层下发子图中的算子打开成小算子，基于小算子组成的子图，进行编译优化，包括buffer fusion、水平融合等，关键是大算子怎样打开、小算子如何重新融等。

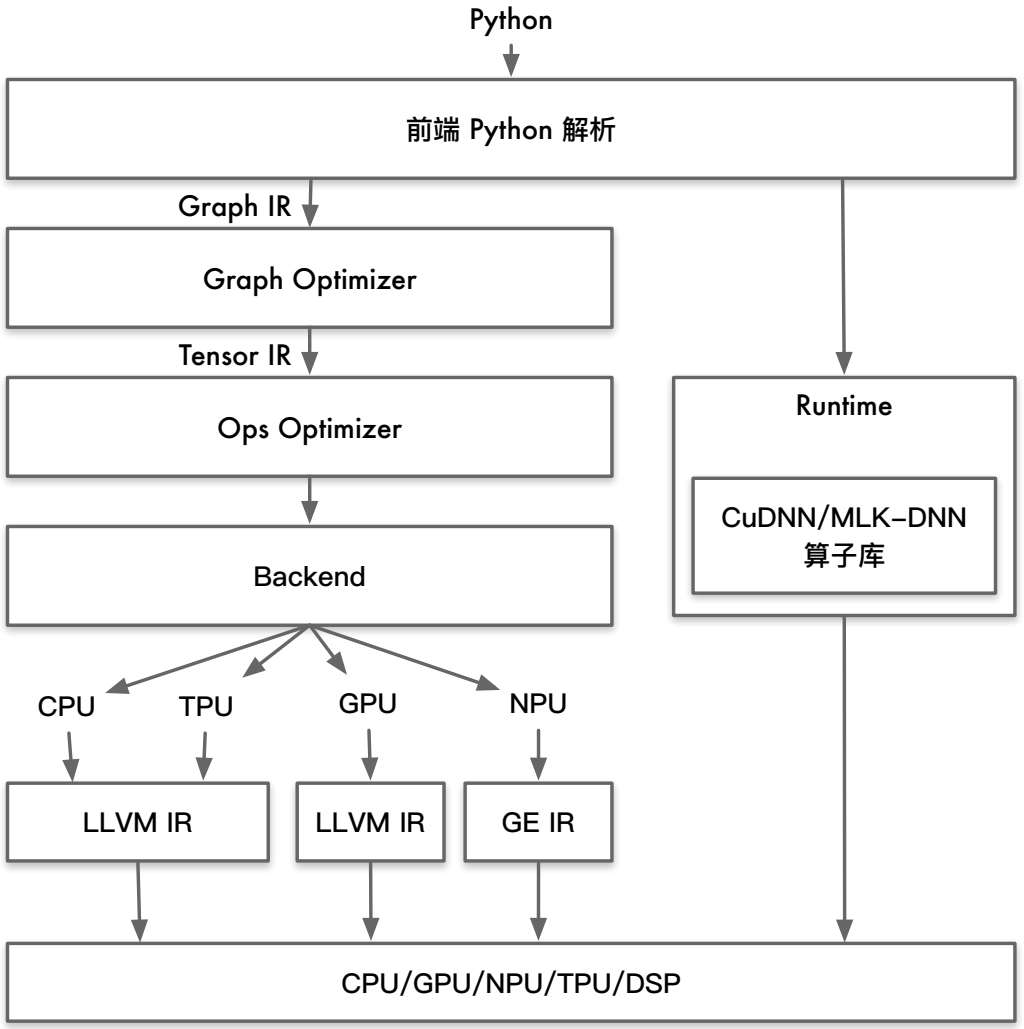
development history: Stage II 专用的AI编译器

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。



development history: Stage II 专用的AI编译器

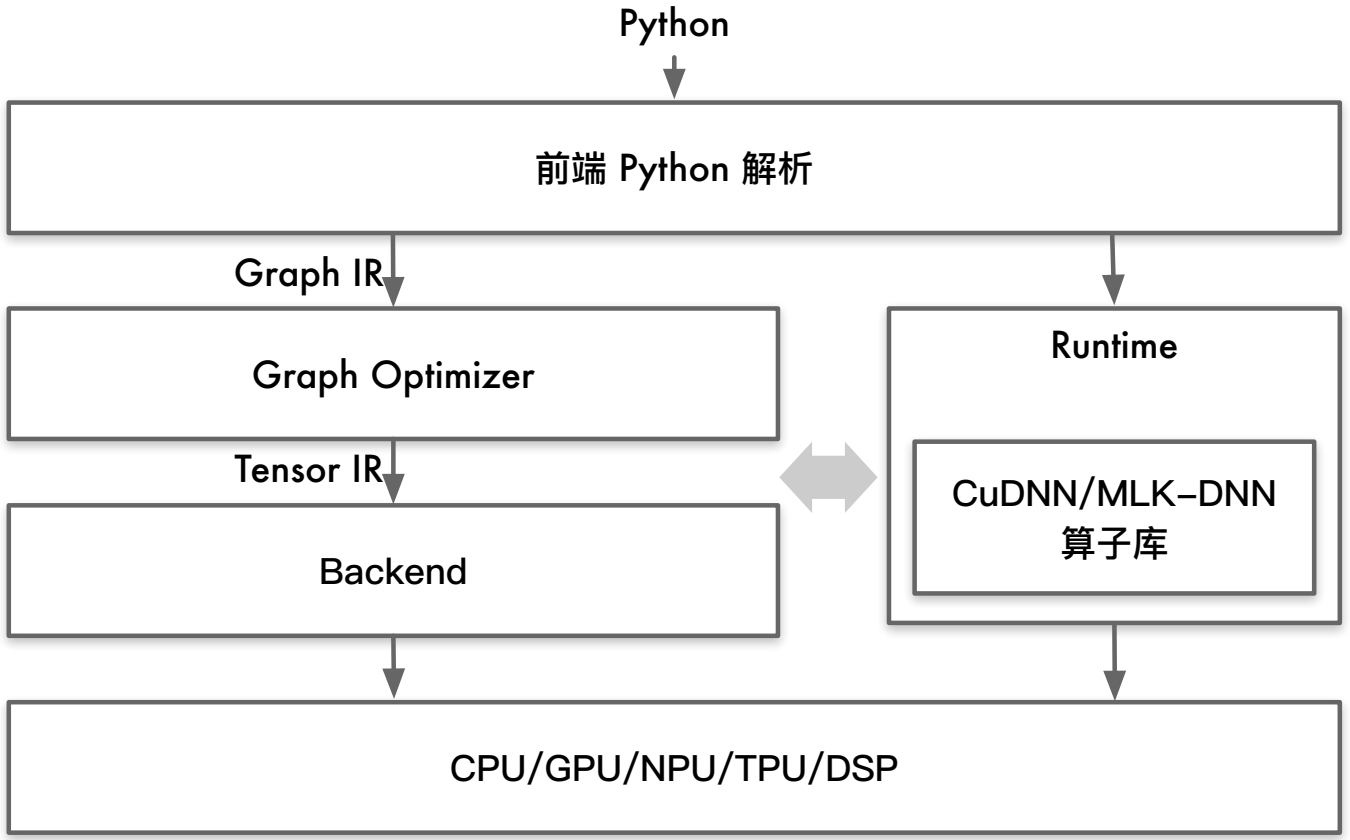
- **表达分离**：计算图层和算子层仍然分开，算法工程师主要关注图层的表达，算子表达和实现主要是框架开发者和芯片厂商提供。
- **功能泛化**：对灵活表达上的动静态图转换、动态 Shape、稀疏计算、分布式并行优化等复杂的需求难以满足。
- **平衡效率和性能**：算子实现上在 Schedule、Tiling、Codegen 上缺乏自动化手段，门槛高，开发者既要了解算子计算逻辑，又要熟悉硬件体系架构。

development history: Stage III 通用AI编译器

- 图算统一表达，实现融合优化
- 算子实现上自动 Schedule、Tiling、Codegen，降低开发门槛
- 更泛化优化能力，实现动静统一、动态 Shape、稀疏性、高阶微分、自动并行等
- 包括编译器、运行时，异构计算、边缘到数据中心都模块化表示和组合，并专注于可用性

What is AI Compiler?

- 图算统一表达，实现融合优化。
- 算子自动生成，降低开发门槛。
- 针对神经网络，泛化优化能力。
- 模块化表示和组合，提升可用性。



At what stage?

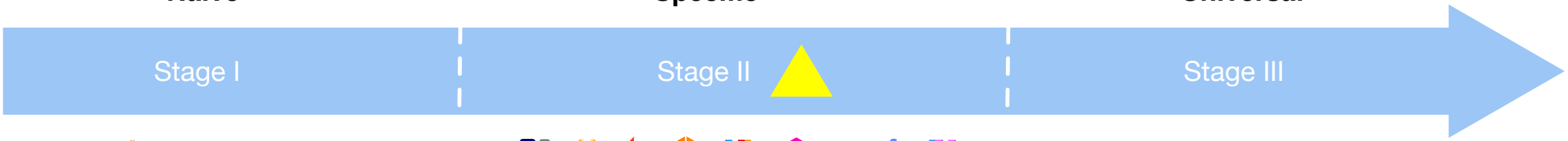
1. 以计算图和算子抽象为主

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

Naive | **Specific** | **Universal**



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

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

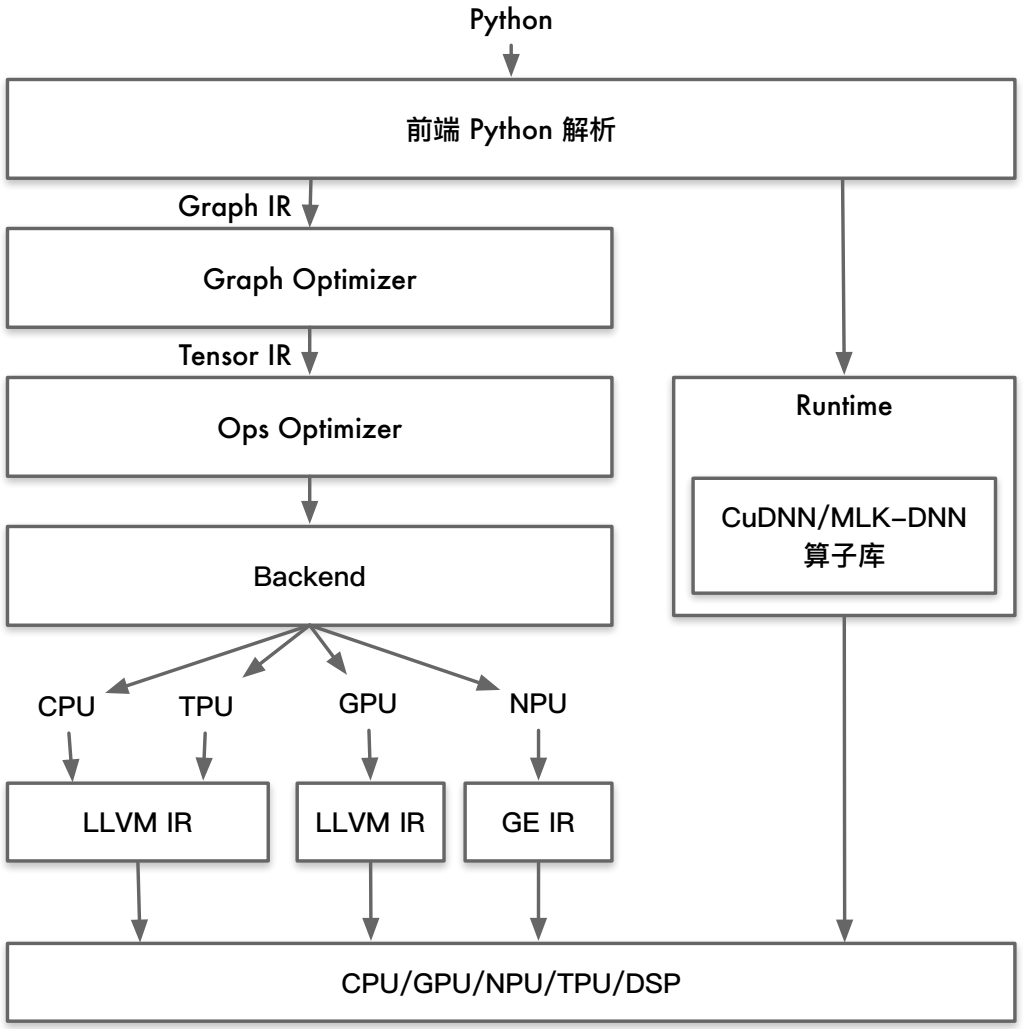
development history: Stage II 专用的AI编译器

表达上：

- 以 PyTorch 为标杆的表达转换到计算图层 IR 进行优化。

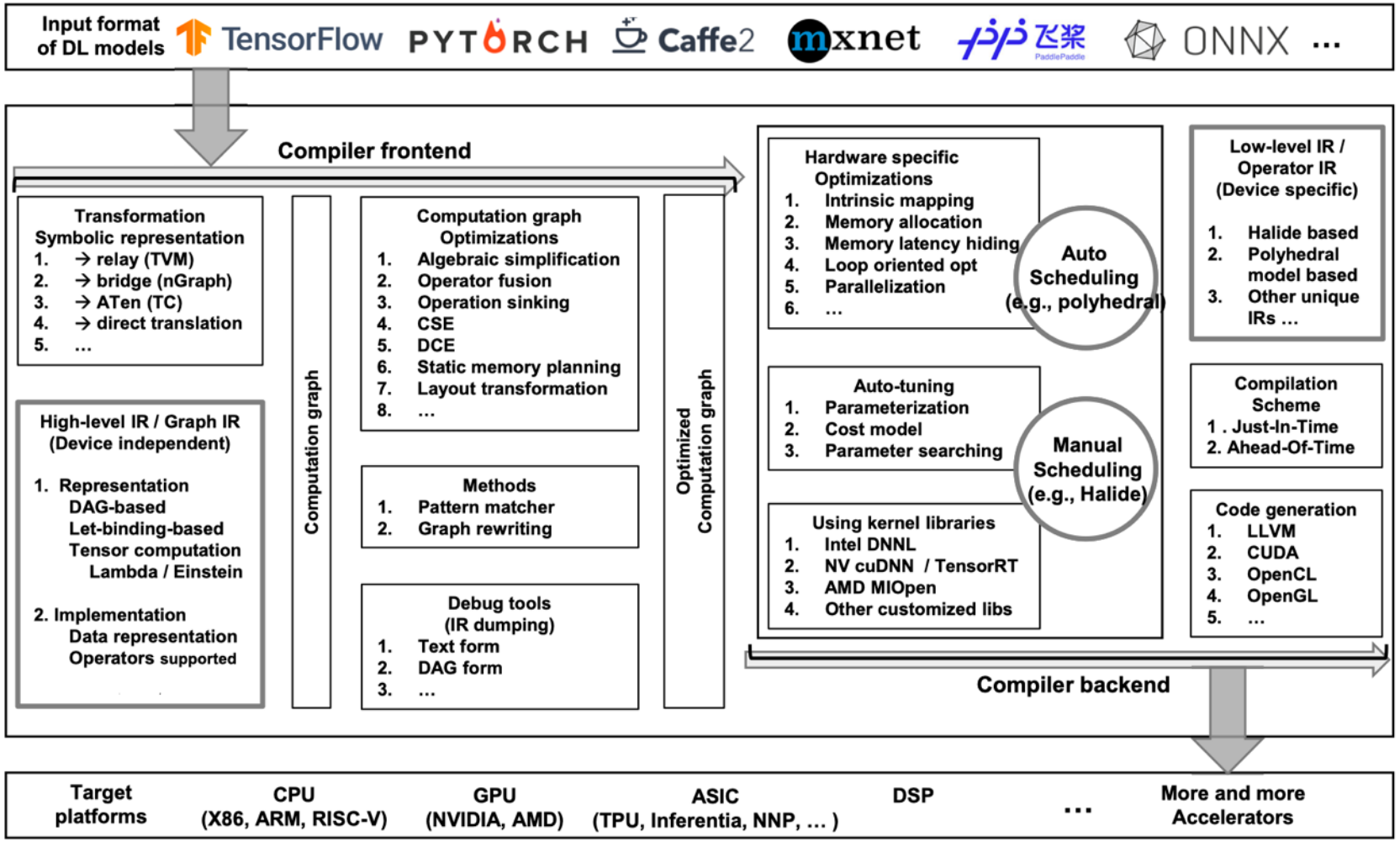
性能上：

- 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。

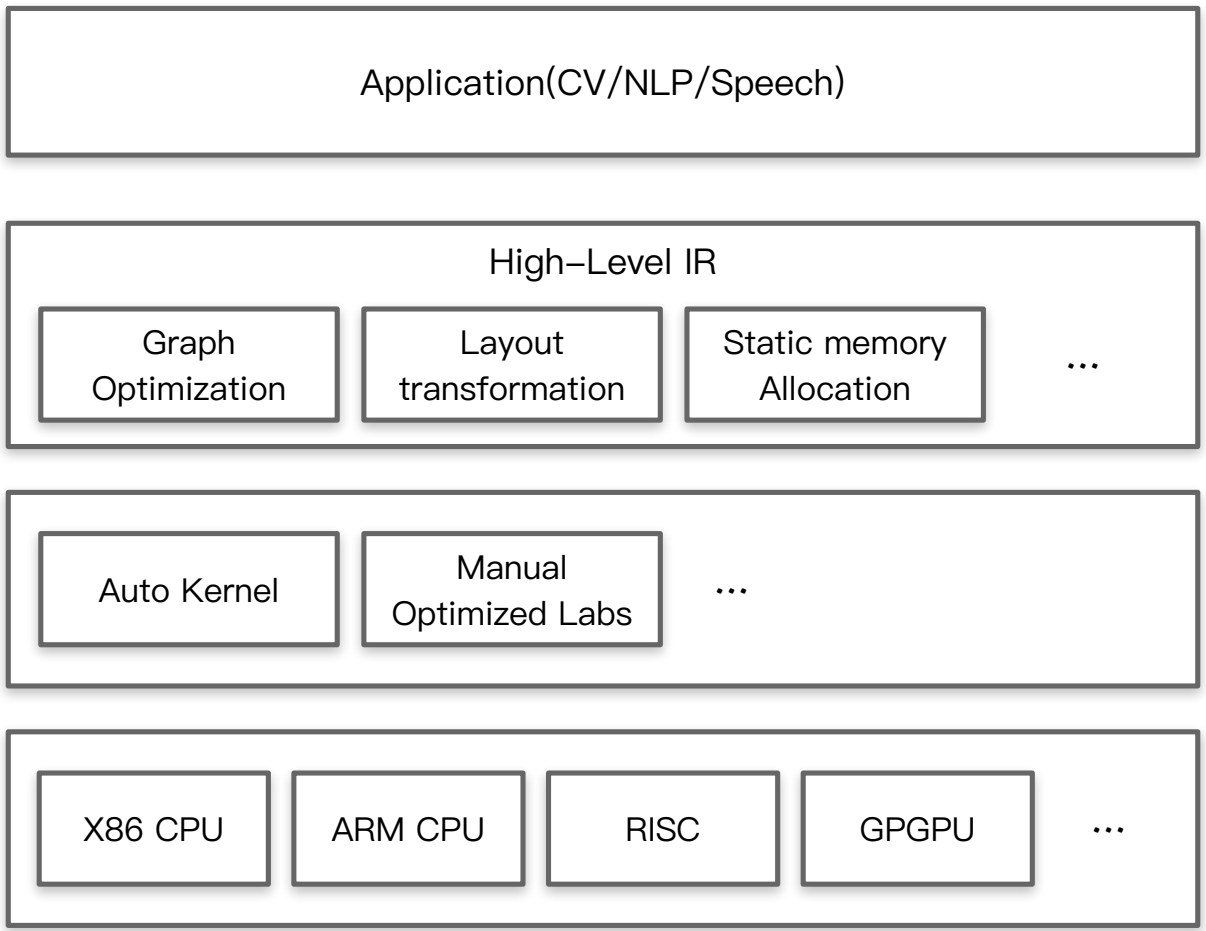
自动调整

- 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

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

Naive | **Specific** | **Universal**

Stage I | Stage II | Stage III

Caffe TensorFlow

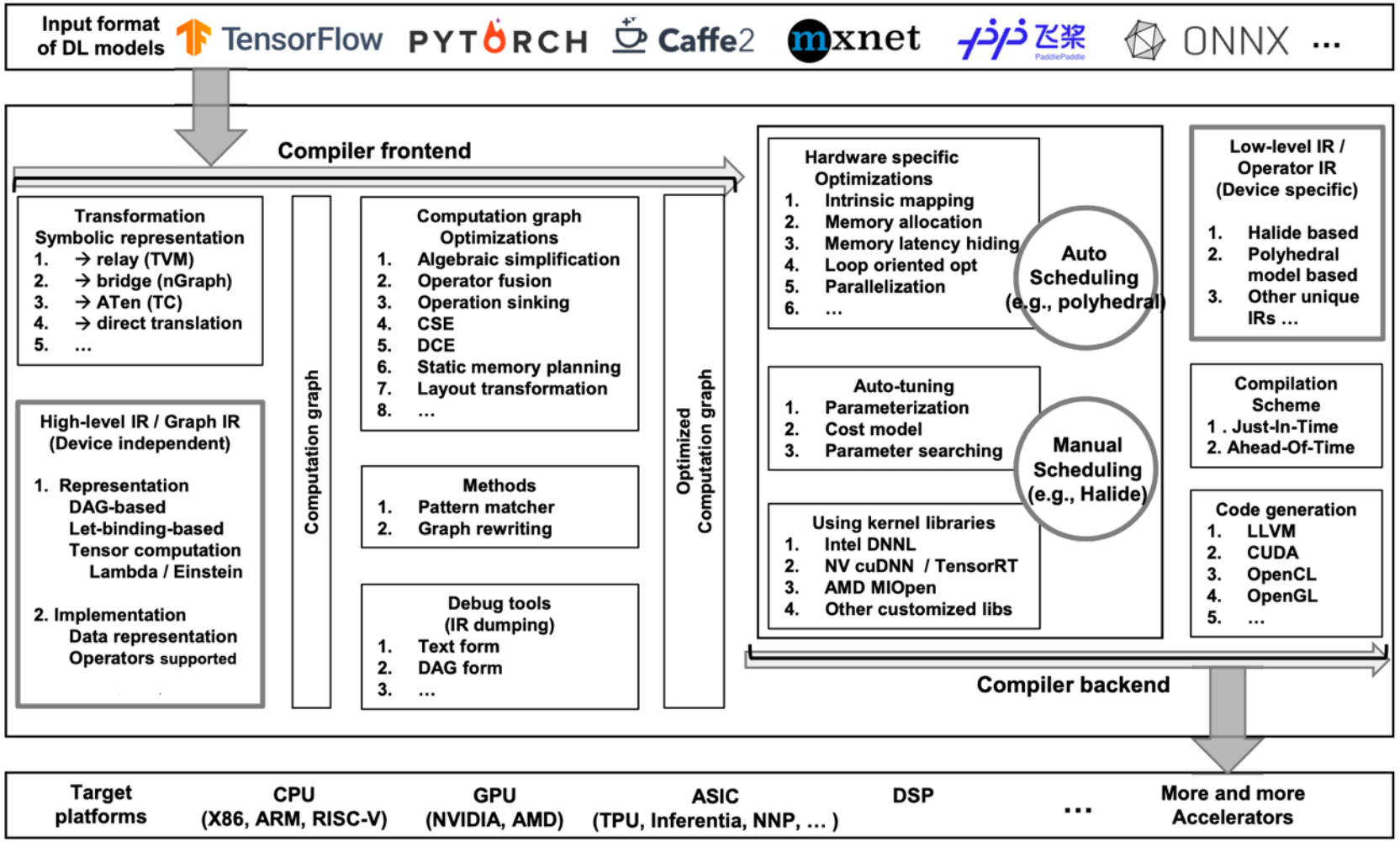
Cons

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