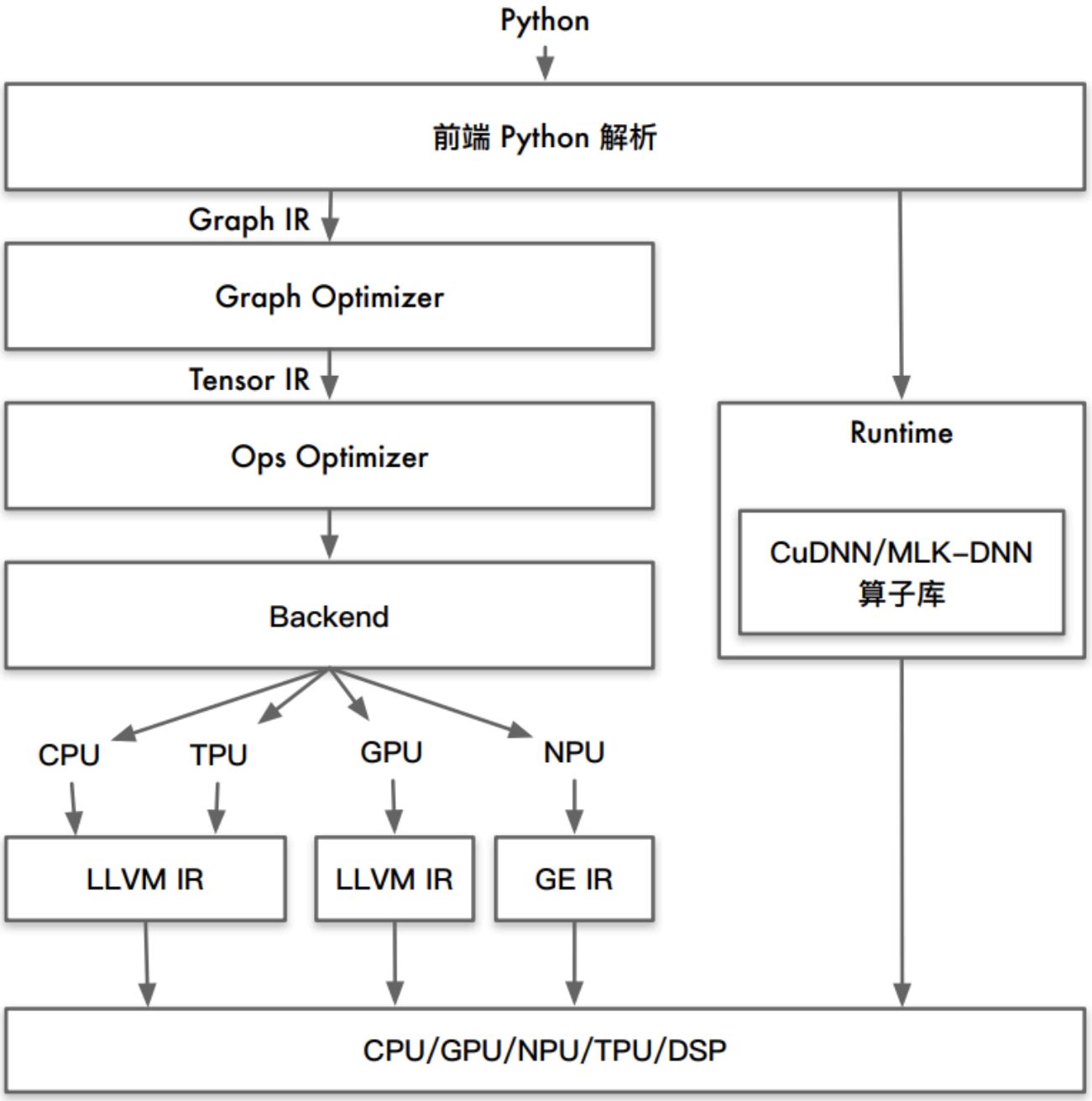
AI编译器

What is AI Compiler

1. 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
2. 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



与传统编译器相比，AI编译器是一个领域特定的编译器，有四个明显的特征：

1、Python为主的动态解释器语言前端

与传统编译器不同，AI编译器通常不需要Lexer/Parser，而是基于前端语言（主要是Python）的AST将模型解析并构造为计算图IR，侧重于保留shape、layout等Tensor计算特征信息，当然部分编译器还能保留控制流的信息。

这里的难点在于，Python是一种灵活度极高的解释执行的语言，AI编译器需要把它转到静态的IR上。

2、多层IR设计：

为什么需要多层IR设计，主要是为了同时满足易用性与高性能这两类需求。为了让开发者使用方便，框架前端(图层)会尽量对Tensor计算进行抽象封装，开发者只要关注逻辑意义上的模型和算子；而在后端算子性能优化时，又可以打破算子的边界，从更细粒度的循环调度等维度，结合不同的硬件特点完成优化。

图编译器

如MindSpore的MindCompiler（MindIR）、TF的XLA（HLO），TVM的Relay等，重点关注非循环相关的优化。除了传统编译器中常见的常量折叠、代数化简、公共子表达式等优化外，还会完成Layout转换，算子融合等优化，通过分析和优化现有网络计算图逻辑，对原有计算逻辑进行拆分、重组、融合等操作，以减少算子执行间隙的开销并且提升设备计算资源利用率，从而实现网络整体执行时间的优化。

算子编译器：

如MindSpore AKG、CANN TBE、TVM(HalideIR)等。针对Low-level IR主要有循环变换、循环切分等调度相关的优化，与硬件intrinsic映射、内存分配等后端pass优化。其中，当前的自动调度优化主要包含了基于搜索的自动调度优化（如ansor[8]）和基于polyhedral编译技术的自动调度优化（如TC和MindAKG[9]）

Codegen

当前基本上收敛在LLVM上。

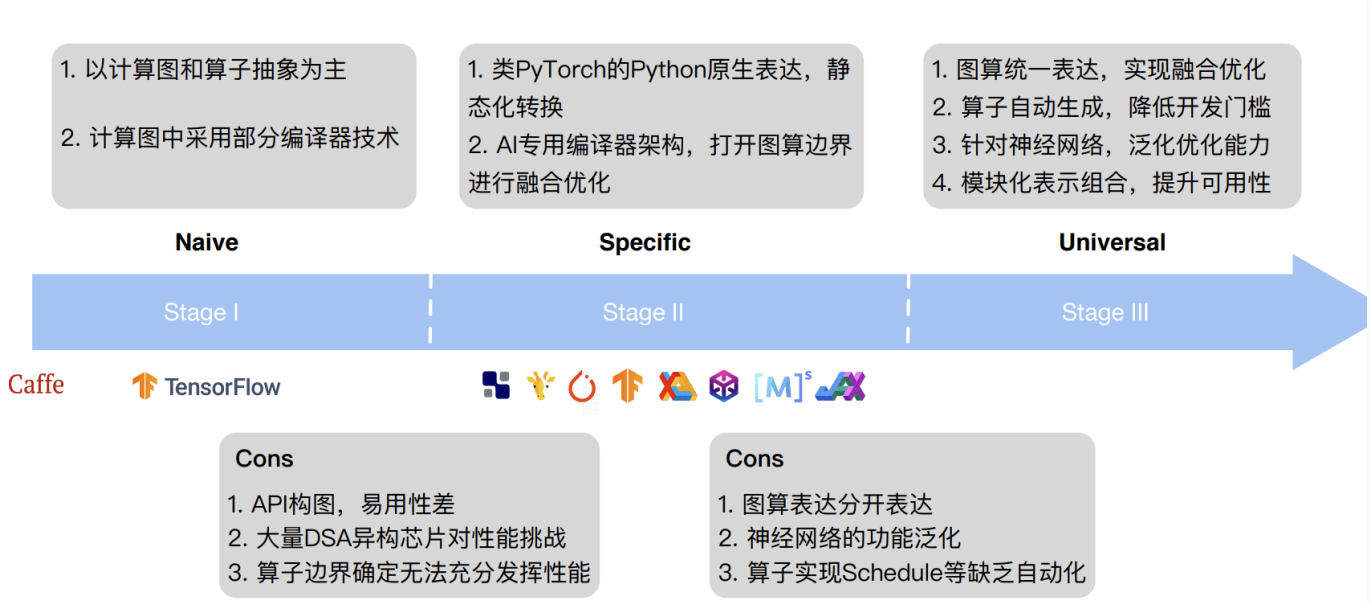
3、面向神经网络的特定优化

数据类型-Tensor：AI领域，计算被抽象成张量的计算，这就意味着AI编译器中主要处理的数据类型也是张量，这个是非常重要的前提。 自动微分：BP是深度学习/神经网络最有代表的部分，目前相对已经比较成熟，基于计算图的自动微分、基于Tape和运算符重载的自动微分方案、基于source2source的自动微分都是现在主流的方案。 自动并行：随着深度学习的模型规模越来越大，模型的并行优化也成为编译优化的一部分，包括：数据并行、算子级模型并行、Pipeline模型并行、优化器模型并行和重计算等。

4、DSA芯片架构的支持

SIMT、SIMD、Dataflow：AI的训练和推理对性能和时延都非常敏感，所以大量使用加速器进行计算，所以AI编译器其实是以加速器为中心的编译器，这个也是区别于通用编译器的一个特征。

发展历史

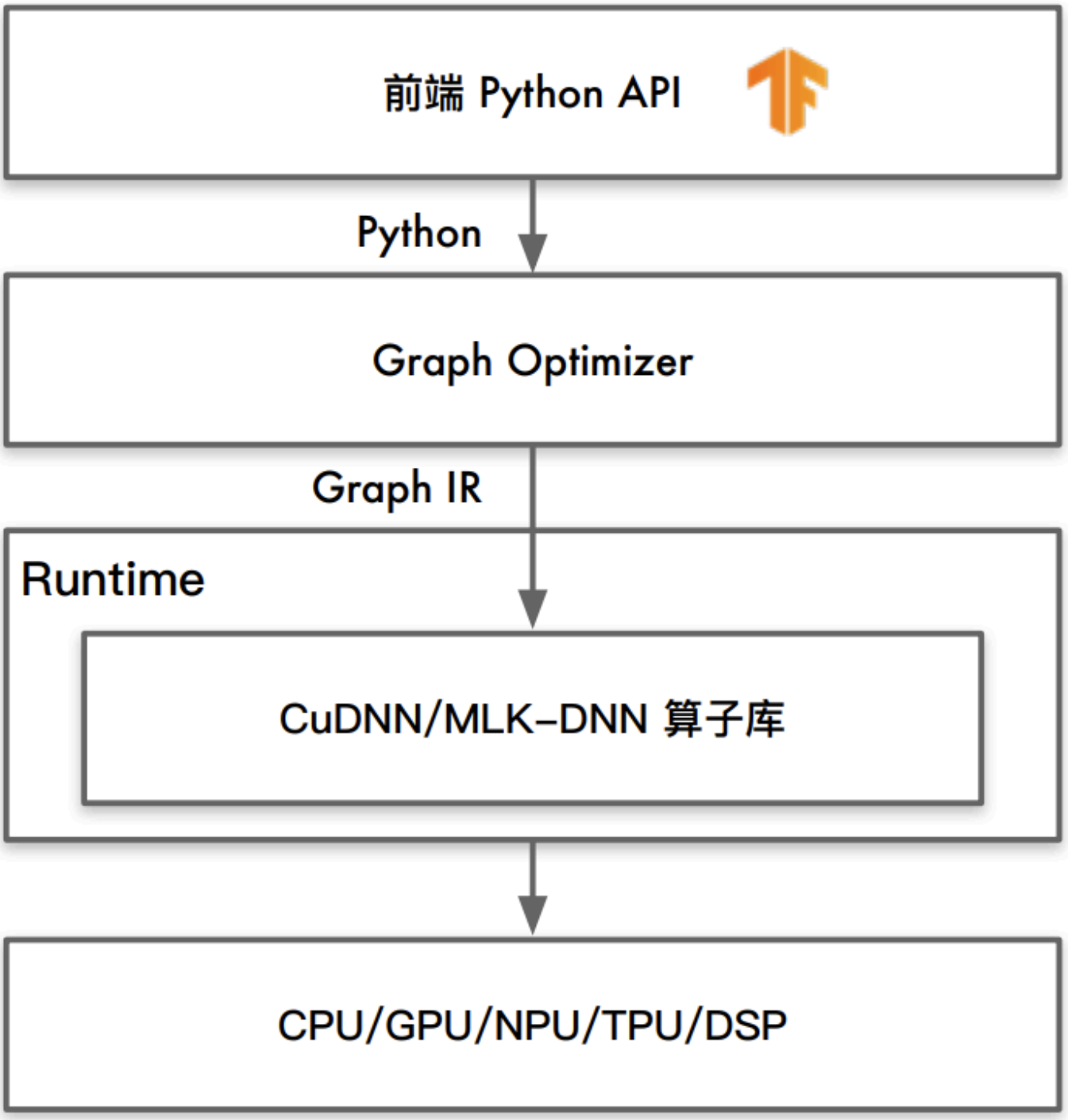


Stage I 朴素的AI编译器

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。

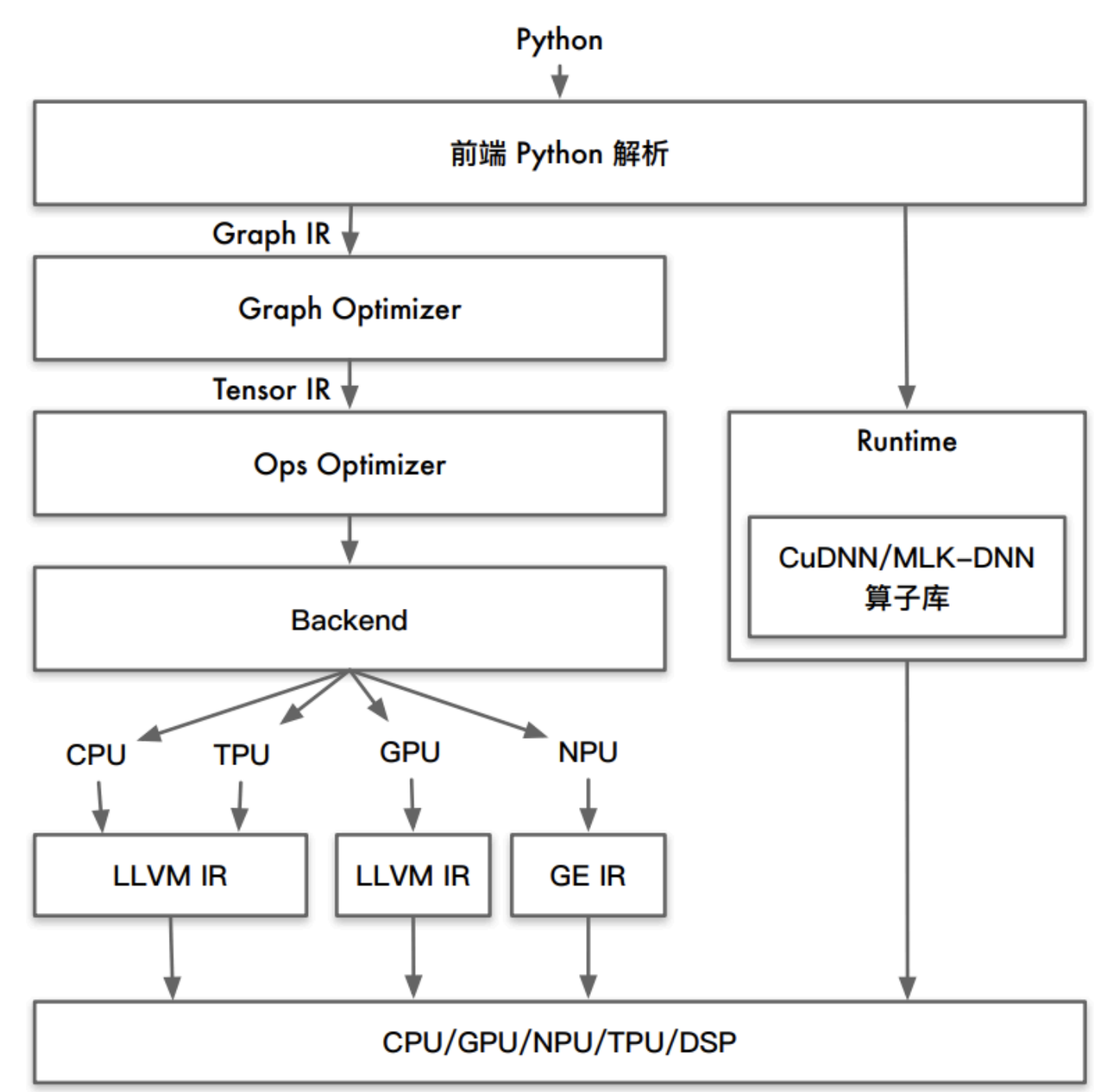
算子层：通常采用手写 kernel 的方式，如在 NVIDIA GPU 上基于 CUDA kernel 实现大量的 .cu 算子或者依赖于 CuDNN 算子优化库



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



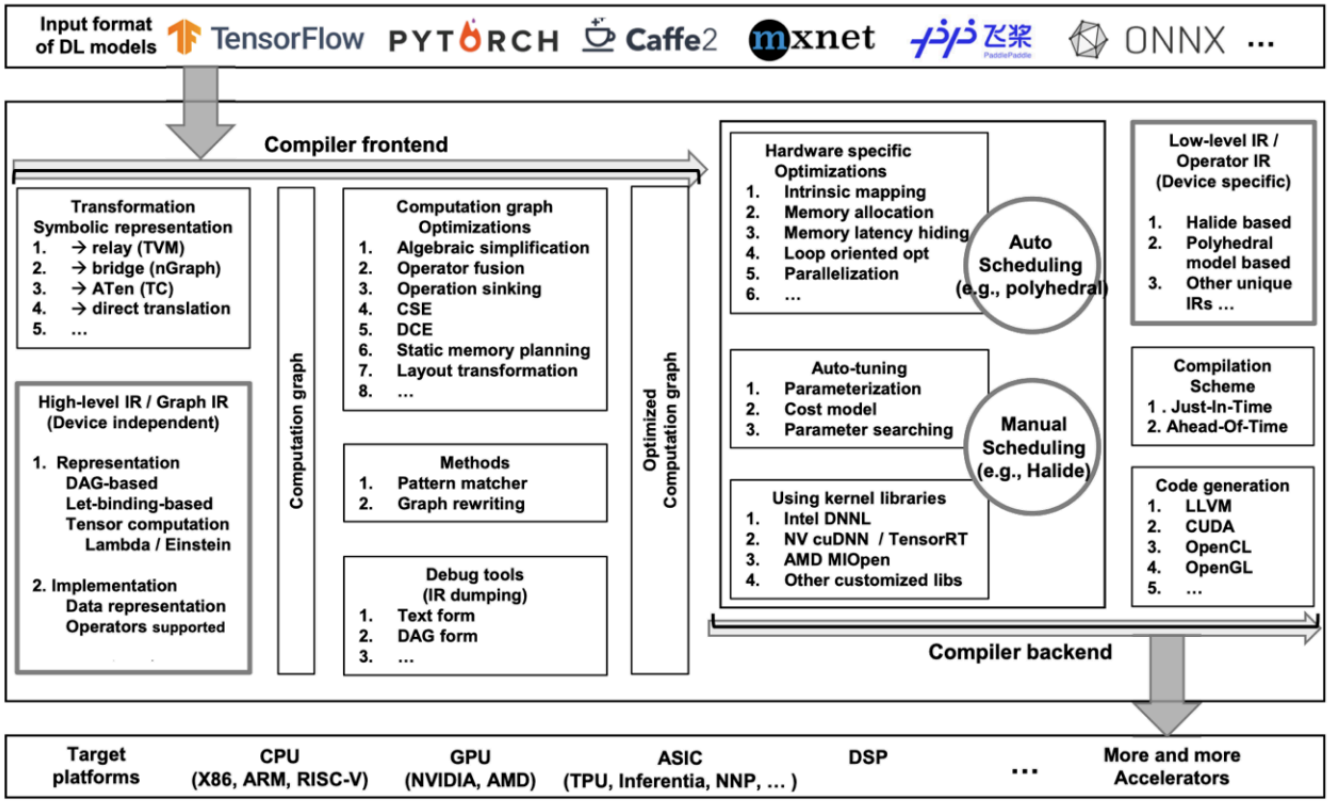
Stage III 通用AI编译器

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

优化分析



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

高层次IR主要用于表示整个深度学习模型的计算图，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。

高层次IR在抽象层次上接近模型的数学描述，能够简洁地表达复杂的运算，隐藏了具体实现细节。它描述了模型中所有操作（如卷积、池化、激活等）及其数据流关系。通过高层次IR，可以进行全局优化，如层融合（fuse layers）、算子重排（operator reordering）等，优化模型的整体性能。

Low-level IR

更细粒度上表示深度学习模型，接近底层硬件的操作。它描述了具体的计算指令和内存操作，细化到单个算子的实现细节。低层次IR能够根据目标硬件平台的特性进行深度优化，如寄存器分配、指令调度、内存布局优化等。可以直接转换为特定硬件平台的高效执行代码

高层次IR与低层次IR的关系

转换过程：高层次IR表示深度学习模型的抽象计算图，经过优化后，转换为低层次IR，以便进行更细粒度的硬件特定优化。

优化路径：高层次IR进行全局优化和算子融合，优化后的IR再转化为低层次IR，在低层次IR上进行寄存器分配、指令调度等底层优化。

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（Dead Code Elimination）

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。

向量化优化

向量化优化是一种编译器技术，旨在利用现代处理器中的向量处理能力（SIMD - 单指令多数据）来提高程序执行效率。

向量化使得多个数据元素能够并行处理，从而加速计算密集型任务，如数值计算、图像处理和科学计算。

SIMD（单指令多数据）：一种并行计算方式，在一条指令下同时对多个数据进行相同的操作

向量化优化的工作原理

1. 识别向量化机会：编译器首先识别代码中可以进行向量化的部分，通常是循环结构。典型的标量循环操作可以改写为向量操作。
2. 数据对齐：确保数据在内存中的对齐方式适合向量化处理，向量化处理器通常要求数据在特定的内存地址对齐。
3. 生成向量指令：将标量操作转换为向量操作。例如，将多个标量加法操作转换为单条向量加法指令。
4. 循环展开：为了提高向量化效果，编译器可能进行循环展开，即将一个循环展开成多个不重复的循环体，从而减少循环控制开销。

示例

```
void add(float *a, float *b, float *c, int n) {
    for (int i = 0; i < n; i++) {
        c[i] = a[i] + b[i];
    }
}
```

```
void add(float *a, float *b, float *c, int n) {
    for (int i = 0; i < n; i += 4) {
        c[i:i+3] = a[i:i+3] + b[i:i+3]; // 每次处理4个元素
    }
}
```

编译器的角色

现代编译器（如GCC、Clang、ICC等）可以自动进行向量化优化。编译器会在编译过程中分析代码结构，特别是循环，判断是否可以安全地进行向量化，然后生成相应的SIMD

指令。

硬件要求

1. SIMD（单指令多数据）架构
2. 寄存器和数据对齐 寄存器宽度：向量寄存器的宽度决定了每次可以处理的数据量。例如，128位寄存器可以一次操作4个32位浮点数或8个16位整数。 数据对齐：向量化操作要求数据在内存中的地址对齐，以匹配寄存器宽度。未对齐的数据会导致额外的开销和性能下降。
3. 内存带宽和缓存 内存带宽：高效的向量化操作需要高内存带宽，以确保数据能够快速加载到寄存器中。 缓存优化：处理器缓存对向量化性能至关重要，编译器和程序员需要优化数据访问模式，以最大限度地利用缓存。
4. 硬件的发展趋势 更宽的向量寄存器：如Intel AVX-512和ARM SVE，支持更宽的向量寄存器，提高并行处理能力。 专用加速器：如TPU（Tensor Processing Unit）、DPU（Data Processing Unit），专门用于加速特定类型的计算任务，进一步提升性能。