AI编译器系列

LLVM架构和原理

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 – 与传统编译器的区别，未来的挑战与思考

Talk Overview

LLVM/Clang process and principle – LLVM 架构和原理

- LLVM 项目发展历史
- LLVM 基本设计原则和架构
- LLVM 中间表示 LLVM IR
- LLVM 前端过程
- LLVM 中间优化
- LLVM 后端生成
- 基于 LLVM 项目

Why U Love Compiler not AI?

【MindSpore进阶】21 训练与评估-保存导出

28 8-19

【MindSpore进阶】19 训练与评估-Model使用

50 8-19

【MindSpore进阶】13 网络构造-构建网络

58 8-19

【MindSpore进阶】16 网络构建-总结

39 8-19

【MindSpore进阶】15 网络构建-控制流

25 8-19

【MindSpore进阶】06 数据处理-格式转换

47 8-19

【MindSpore进阶】14 网络构建-自动求导

31 8-19

【MindSpore进阶】12 网络构造-优化器

42 8-19

【MindSpore进阶】05 数据处理-数据迭代

45 8-19

【MindSpore进阶】17 训练与评估-评估指标

46 8-19

<100

Why U Love Compiler not AI?

【计算图】第六篇！计算图未来将会走向何方?

188 10-12

【计算图】第五篇！PyTorch控制流如何实现？动静统一原

270 10-11

【计算图】第四篇！图优化与执行调度！计算图优化！单算

227 10-10

【计算图】系列第三篇！计算图跟微分什么关系？怎么用计

261 10-9

【计算图】系列第一篇！计算图有哪些内容知识?

251 10-6

【AI框架基础】系列第四篇！函数式编程和声明式编程有什

129 10-6

【AI框架基础】系列第三篇！AI框架之争！都2022年，还在

175 10-5

【AI框架基础】系列第二篇！AI框架有什么用？没有AI框架

188 10-5

300+

Why U Love Compiler not AI?

什么是张量并行？张量并行的数学原理是什么？【大模型与

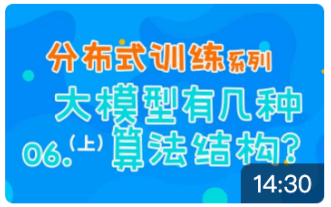
126 11-5

PyTorch数据并行怎么实现？DP、DDP、FSDP数据并行原

271 11-1

从十亿到万亿规模SOTA大模型有哪些？BERT、GPT3、

163 10-30



大模型算法有哪种结构？怎么样才算大模型呢？从

212 10-29

AI集群机器间是怎么通信的？通信原语是个什么玩意？【大

149 10-27

如何实现集群的通信？硬件PCIE、NVLINK、RDMA原

460 10-27

AI集群用哪种服务器架构？Ring All Reduce算法跟物理

174 10-25

AI框架如何实现分布式训练功能的？SISD和SIMT又是什

194 10-25

大模型是什么？大模型有什么用？训练大模型会遇到哪些挑

256 10-15

大规模分布式训练的研究内容！【大模型与分布式训练】

266 10-15

<600

Why U Love Compiler not AI?

LLVM编译器前端和优化层了解下？词法语法分析、Pass优

10 1小时前

LLVM IR详解！LLVM编译器的核心理念来啦！【AI编译

1562 11-23

LLVM架构了解下？为什么

4770 11-22

GCC编译过程！优缺点是啥？

1995 11-19

GCC和LLVM发家历史？两大

6162 11-19

编译器和解释器啥区别？AOT和JIT啥区别？Pass和IR又是

1359 11-15

分布式训练总结！【大模型与分布式训练】系列第十篇

101 11-11

混合并行？多维并行？有多维度混合在一起并行吗？【大模

150 11-6

分布式训练的流水线并行来啦！了解下GPipe和

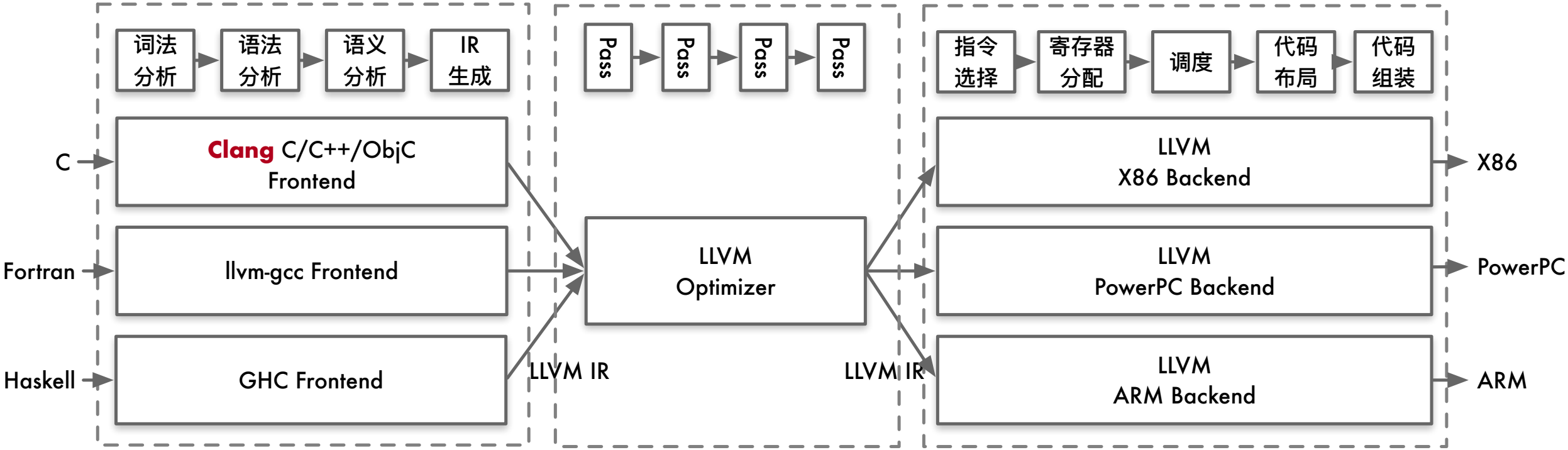
157 11-5

张量还能自动并行？MindSpore张量自动并行啥原

107 11-5

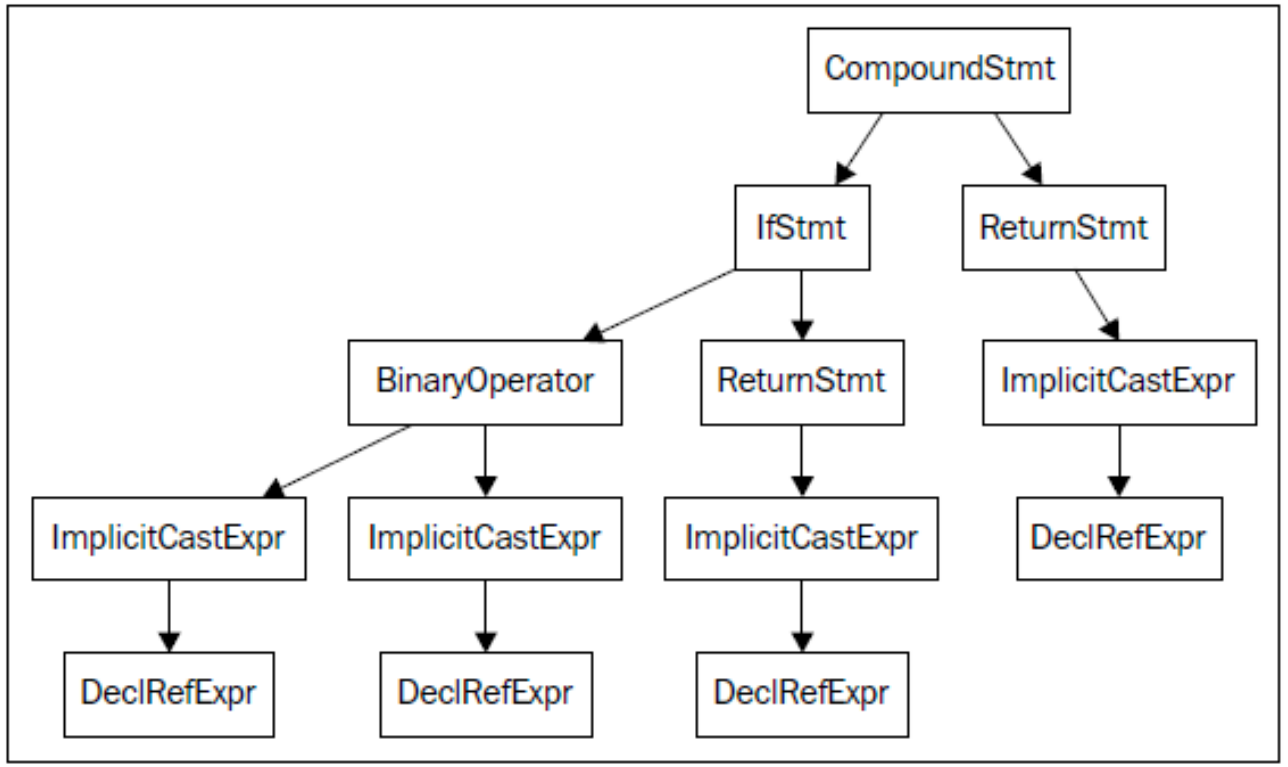
1500

LLVM Architecture



Syntactic analysis 语法分析

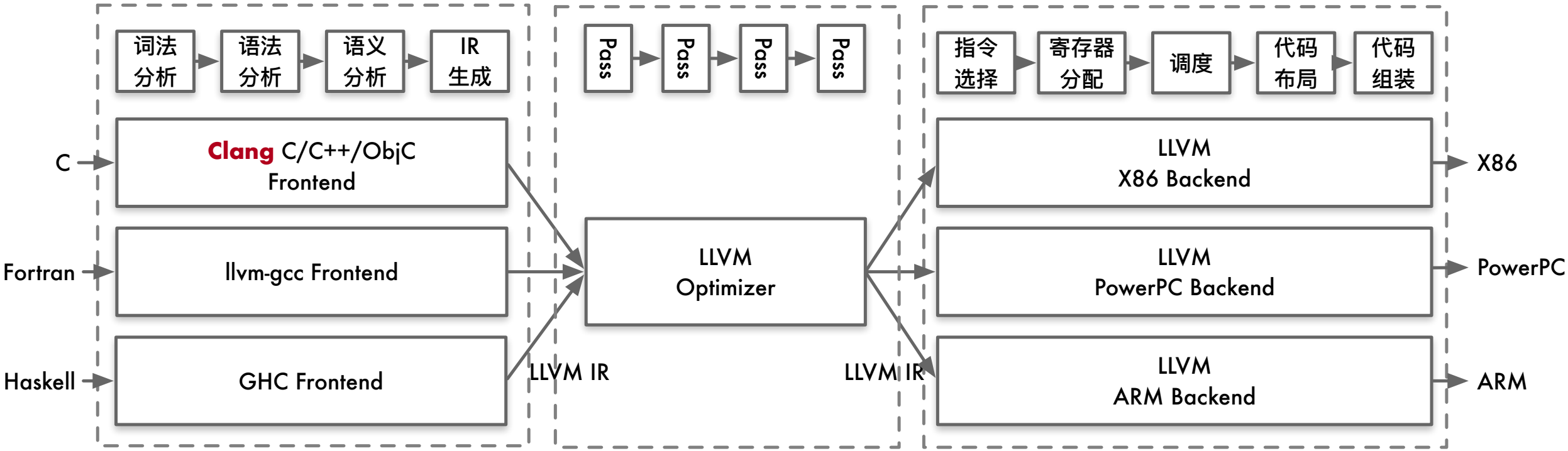
- 分组标记以形成表达式、语句、函数体等。检查一组标记是否有意义，考虑代码物理布局，未分析代码的意思，就像英语中的语法分析，不关心你说了什么，只考虑句子是否正确，并输出语法树（AST）。



$ clang -fsyntax-only -Xclang -ast-dump hello.c

LLVM Architecture

目标无关优化，理解优化操作，实际上就是理解 IR 如何在 pass 流水线中被修改，这需要知道每个 pass 执行的修改，还有各个 pass 是以什么顺序被执行。



Finding Pass

https://llvm.org/docs/Passes.html

优化通常由分析 Pass 和转换 Pass 组成。

- **分析 Pass**：负责发掘性质和优化机会；
- **转换 Pass**：生成必需的数据结构，后续为后者所用；

- -adce: Aggressive Dead Code Elimination

积极的死代码消除。此pass类似于DCE，但它假定值是死的，除非得到其他证明。这类似于SCCP，除了用于值的活动性。

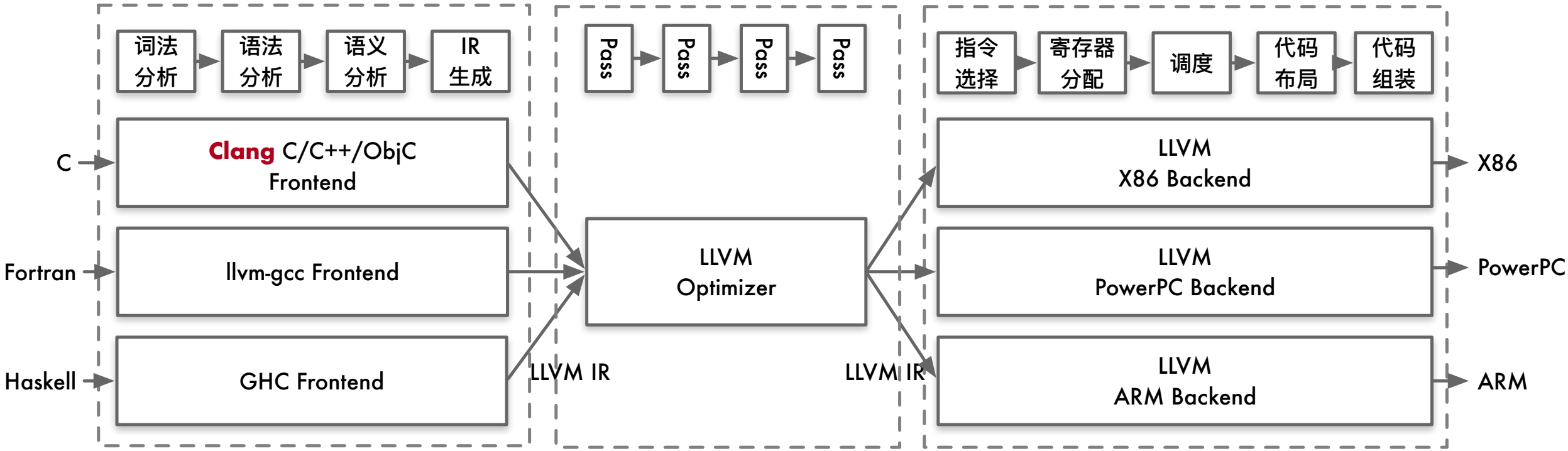
- -constmerge: Merge Duplicate Global Constants

将重复的全局常量合并到一个共享的常量中。这是有用的，一些passes在程序中插入许多字符串常量，不管现有字符串是否可用。

LLVM 后端

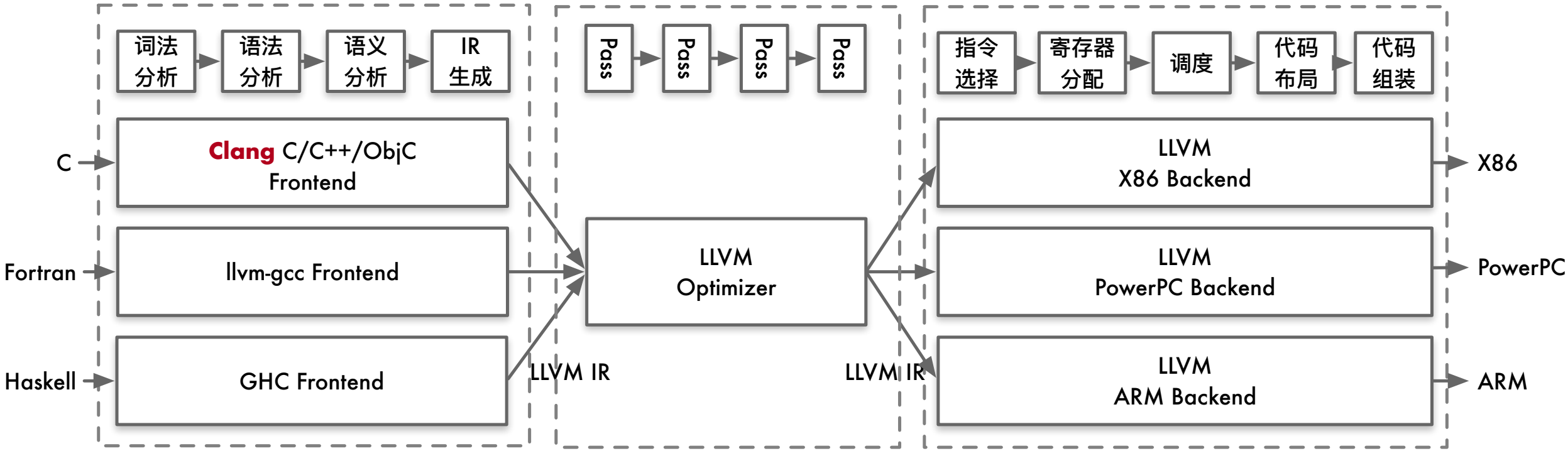
LLVM Architecture

后端由一套分析和转换 Pass 组成，它们的任务是代码生成，即将LLVM IR变换为目标代码（或者汇编）



LLVM Architecture

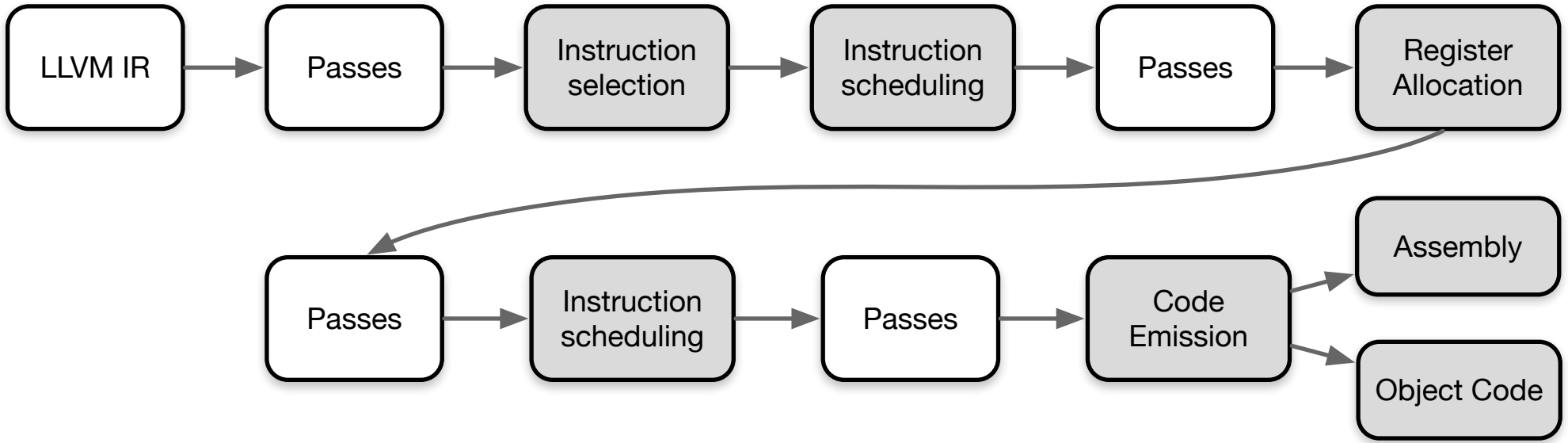
后端由一套分析和转换 Pass 组成，它们的任务是代码生成，即将LLVM IR变换为目标代码（或者汇编）



LLVM Backend Pass

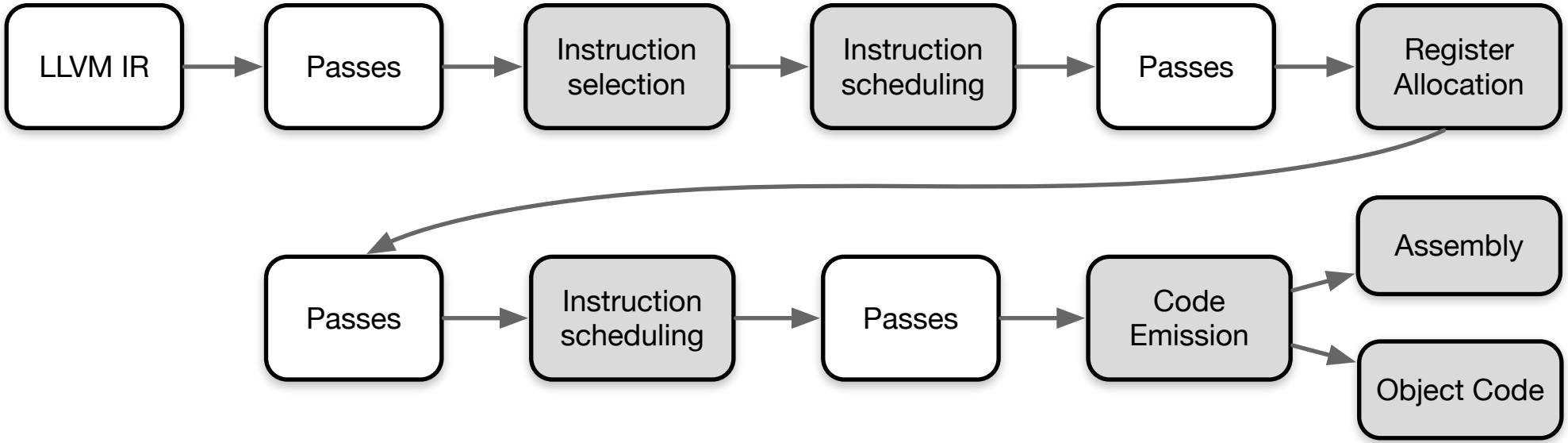
整个后端流水线用到了四种不同层次的指令表示：

- 内存中的LLVM IR，SelectionDAG 节点，MachineInstr，和 MCInst。



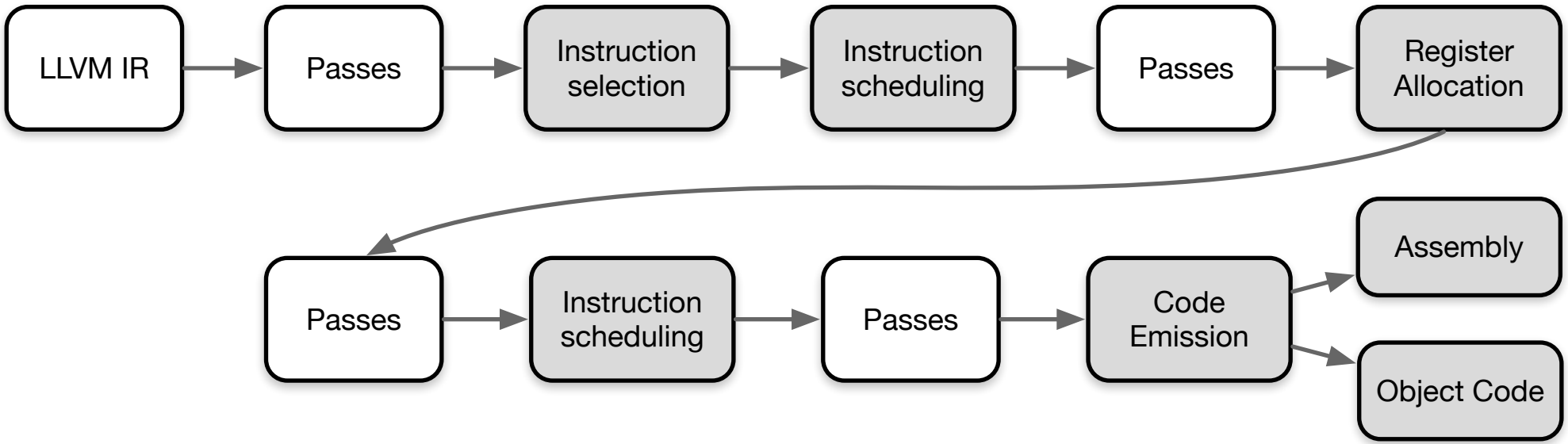
Instruction Selection 指令选择

- 内存中 LLVM IR 变换为目标特定 SelectionDAG 节点；
- 每个DAG能够表示单一基本块的计算；
- 节点表示指令，而边编码了指令间的数据流依赖；
- **让LLVM代码生成程序库能够运用基于树的模式匹配指令选择算法。**



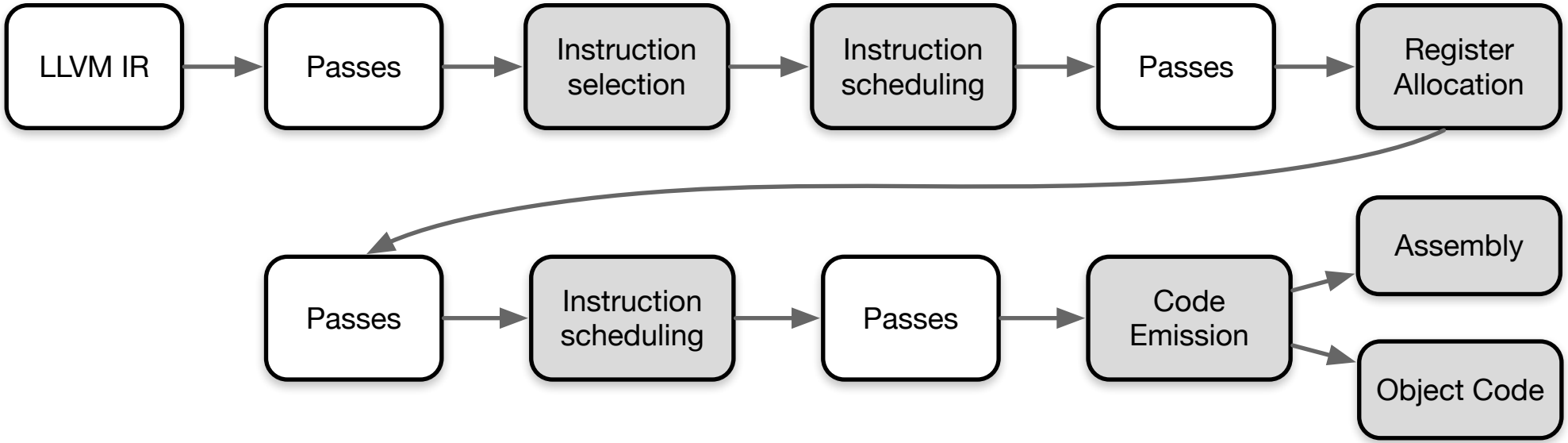
Instruction Scheduling 指令调度

- 第1次指令调度（Instruction Scheduling），也称为前寄存器分配（RA）调度；
- 对指令排序，同时尝试发现尽可能多的指令层次的并行；
- 然后指令被变换为MachineInstr三地址表示。



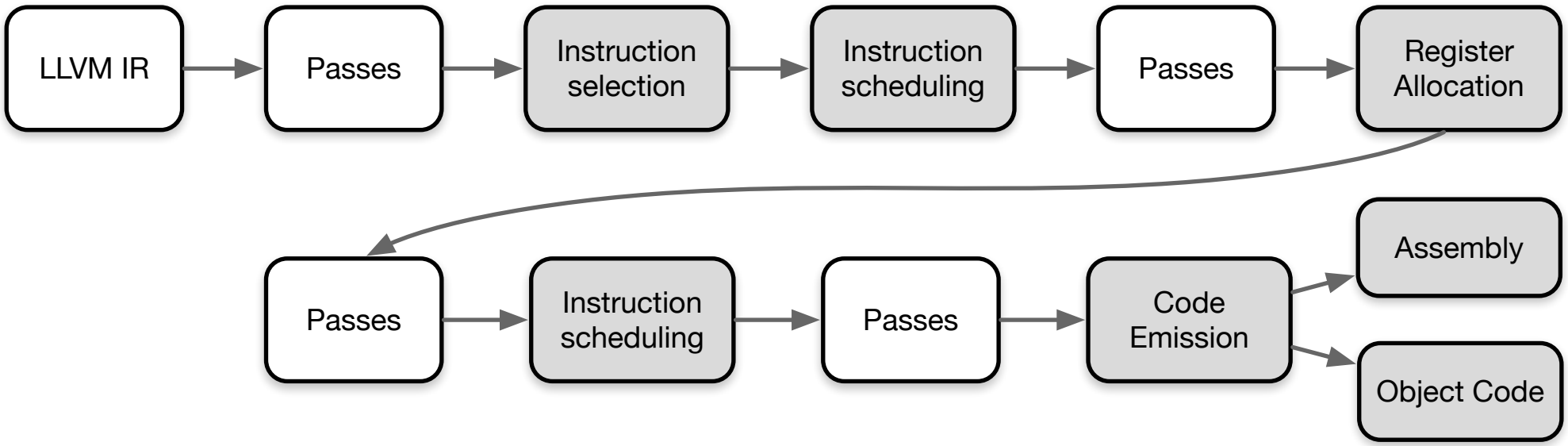
Register Allocation 寄存器分配

- LLVMIR 两个中哟特性之一：LLVM IR 寄存器集是无限；
- 这个性质一直保持着，直到寄存器分配（Register Allocation）；
- 寄存器分配将无限的虚拟寄存器引用转换为有限的目标特定的寄存器集；
- 寄存器不够时挤出（spill）到内存。



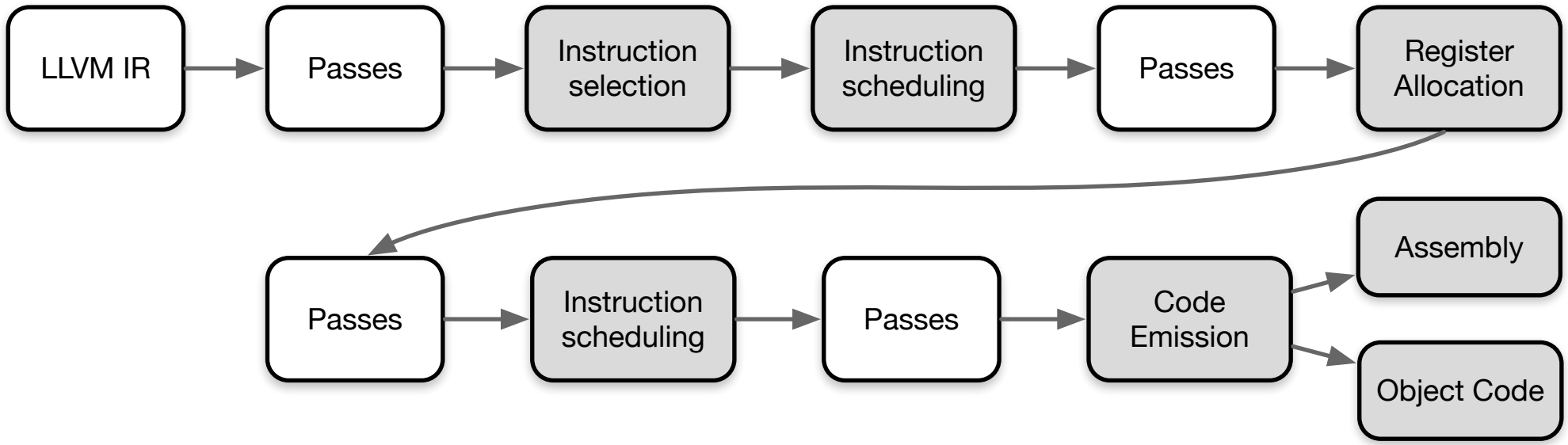
Instruction Scheduling 指令调度

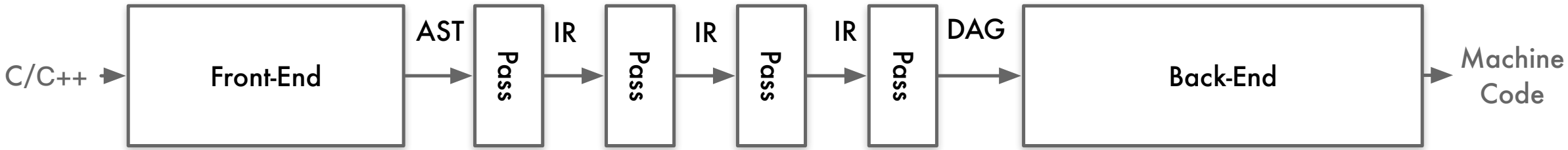
- 第2次指令调度，也称为后寄存器分配（RA）调度；
- 此时可获得真实的寄存器信息，某些类型寄存器存在延迟，它们可被用以改进指令顺序。



Code Emission 代码输出

- 代码输出阶段将指令从 MachineInstr 表示变换为 MCInst 实例；
- 新的表示更适合汇编器和链接器，可以输出汇编代码或者输出二进制块特定目标代码格式。






```c
#include <stdio.h>

int loop_add(int* data, int num) {
  int res = 0;
  for (int i = 0; i < num; i ++) {
    res += data[i];
  }
  return res;
}
```

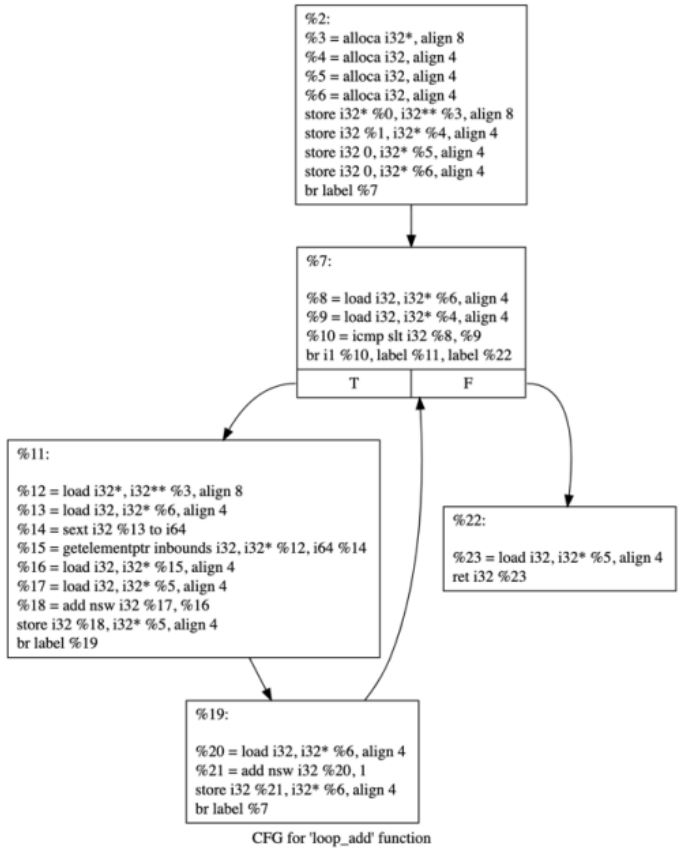
```
define dso_local i32 @loop_add(i32*, i32) #0 {
  %3 = alloca i32*, align 8
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  %6 = alloca i32, align 4
  store i32* %0, i32** %3, align 8
  store i32 %1, i32* %4, align 4
  store i32 0, i32* %5, align 4
  store i32 0, i32* %6, align 4
  br label %7

; <label>:7:                                      ; preds = %19, %2
  %8 = load i32, i32* %6, align 4
  %9 = load i32, i32* %4, align 4
  %10 = icmp slt i32 %8, %9
  br i1 %10, label %11, label %22

; <label>:11:                                     ; preds = %7
  %12 = load i32*, i32** %3, align 8
  %13 = load i32, i32* %6, align 4
  %14 = sext i32 %13 to i64
  %15 = getelementptr inbounds i32, i32* %12, i64 %14
  %16 = load i32, i32* %15, align 4
  %17 = load i32, i32* %5, align 4
  %18 = add nsw i32 %17, %16
  store i32 %18, i32* %5, align 4
  br label %19

; <label>:19:                                     ; preds = %11
  %20 = load i32, i32* %6, align 4
  %21 = add nsw i32 %20, 1
  store i32 %21, i32* %6, align 4
  br label %7

; <label>:22:                                     ; preds = %7
  %23 = load i32, i32* %5, align 4
  ret i32 %23
}
```



CFG for 'loop_add' function

基于LLVM的项目

LLVM之父Chris Lattner：编译器的黄金时代

ASPLOS Keynote: The Golden Age of Compiler Design in an Era of HW/SW Co-design by Dr. Chris Lattner

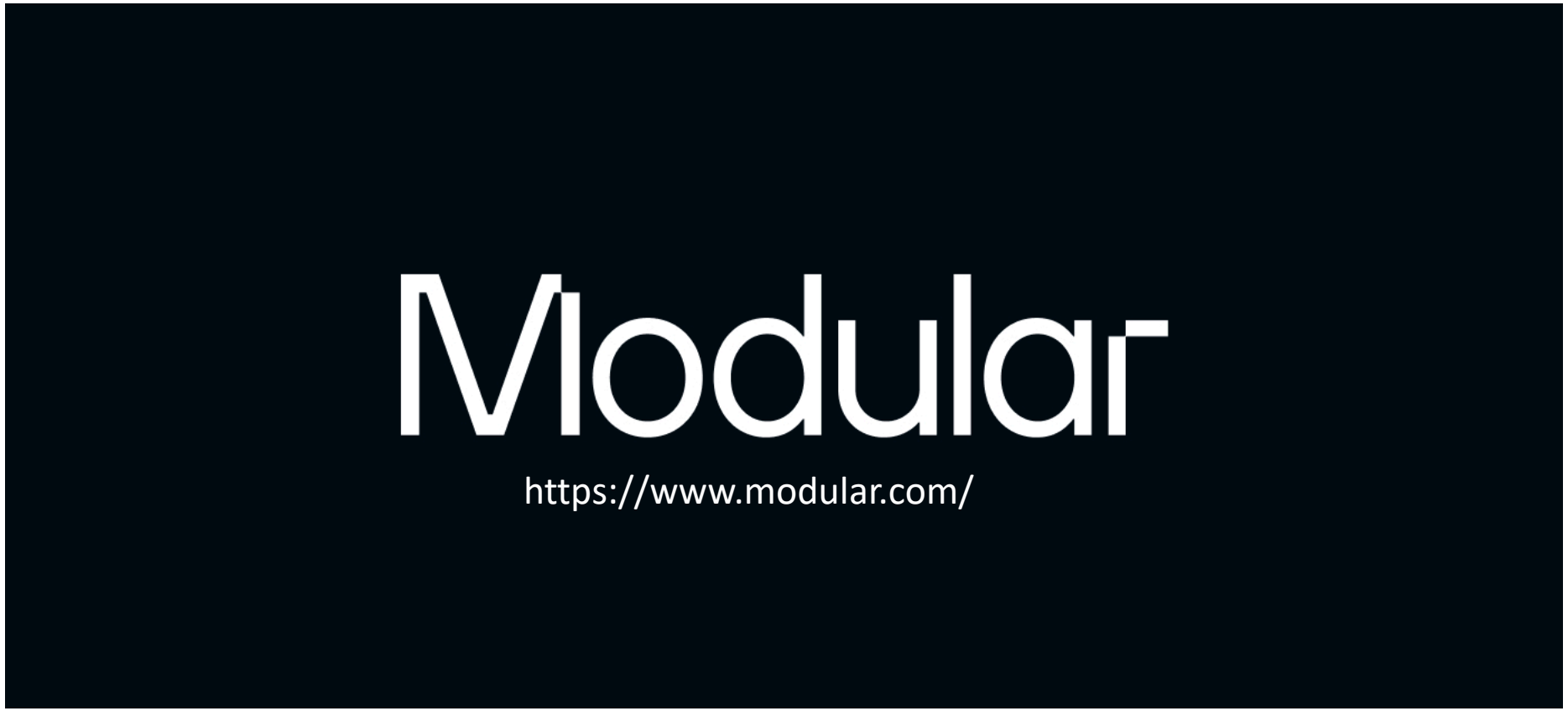
2.7万次观看 · 1年前

SiFiveInc

This week at the ASPLOS 2021 conference, Dr. Chris Lattner gave the keynote address to open the event with a discussion of the ...

A New Golden Age for Computer Architecture John L. Hennessy, David A. Patterson June 2018 End o... 22 个章节

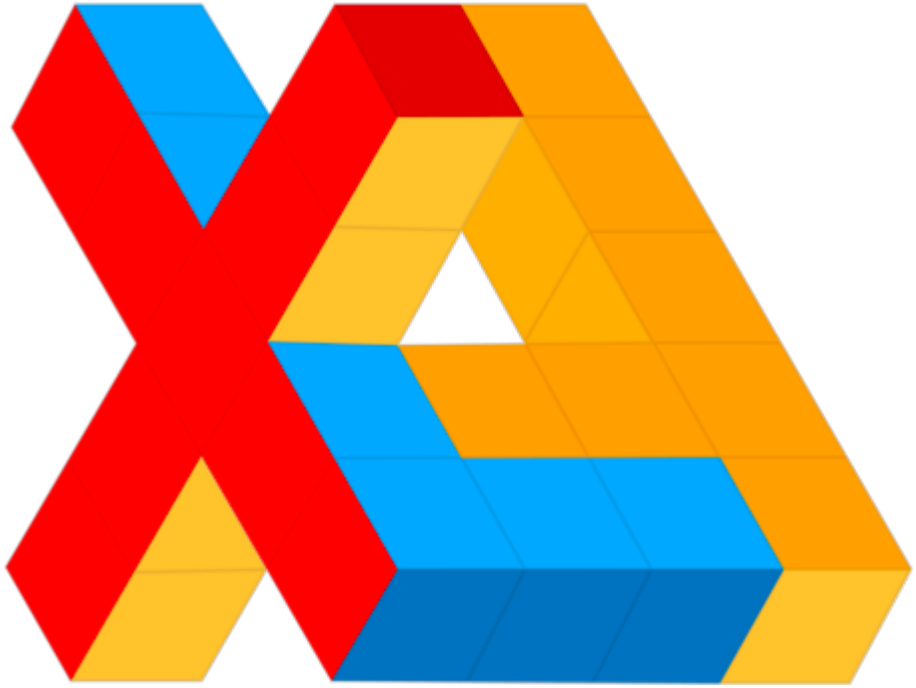
LLVM之父Chris Lattner：编译器的黄金时代



目标是重建全球ML基础设施，包括编译器、运行时，异构计算、边缘到数据中心并重，并专注于可用性，提升开发人员的效率。

XLA：优化机器学习编译器

- XLA（加速线性代数）是一种针对特定领域的线性代数编译器，能够加快 TensorFlow 模型的运行速度，而且可能完全不需要更改源代码。



JAX：高性能的数值计算库

- JAX 是Autograd和XLA的结合,JAX 本身不是一个深度学习的框架,他是一个高性能的数值计算库,更是结合了可组合的函数转换库,用于高性能机器学习研究。

TensorFlow：机器学习平台

- TensorFlow是一个端到端开源机器学习平台。它拥有一个全面而灵活的生态系统，其中包含各种工具、库和社区资源，可助力研究人员推动先进机器学习技术。

TVM 端到端深度学习编译器

- 为了使得各种硬件后端的计算图层级和算子层级优化成为可能，TVM 从现有框架中取得 DL 程序的高层级表示，并产生多硬件平台后端上低层级的优化代码，其目标是展示与人工调优的竞争力。

Julia：面向科学计算的高性能动态编程语言

- 计算中，Julia使用 LLVM JIT编译。LLVM JIT 编译器通常不断地分析正在执行的代码，并且识别代码的一部分，使得从编译中获得的性能加速超过编译该代码的性能开销。