AI编译器系列

AI 编译器的前世

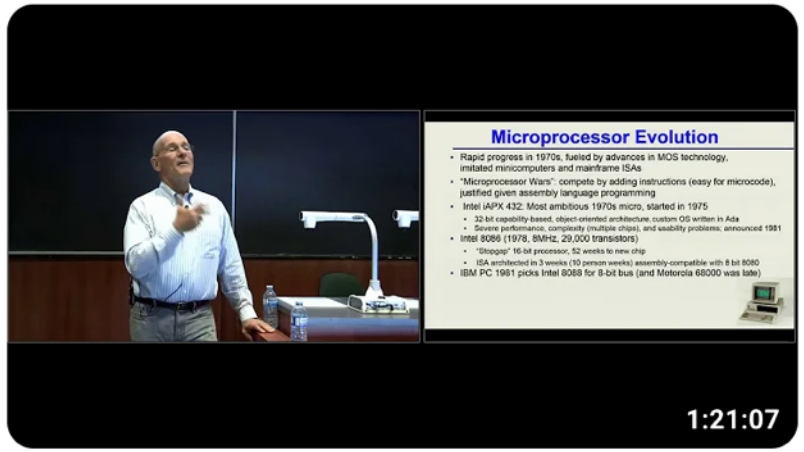
ZOMI

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**
   - Why need AI Compiler – 为什么需要AI编译器
   - Base Common architecture – AI编译器的出现和架构
   - Different and challenge of the future – 未来的挑战与思考

计算机架构的新黄金时代

- A New Golden Age for Computer Architecture: History, Challenges and Opportunities
  https://www.youtube.com/watch?v=kFT54hO1X8M



David Patterson - A New Golden Age for Computer Architecture: History, Challenges and Opportunities

7.1万次观看 · 3年前

UBC Computer Science

Abstract: In the 1980s, Mead and Conway democratized chip design and high-level language programming surpassed assembly ...

Turing Awards | What is Computer Architecture | IBM System360 | Semiconductors | Microprocessor... 44 个章节

编译器的黄金时代

- The Golden Age of Compiler Design in an Era of HW/SW Co-design
- https://www.youtube.com/watch?v=4HgShra-KnY

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

为什么需要 AI 编译器

Hardware is getting harder

- Modern compute acceleration platforms are multi-level and explicit:
  - scalar, Vector, Multi-core, Multi-package, Multi-rack, et. al.
  - None-coherent memory subsystems increase efficiency
- Heterogenous compute incorporating domain-specific accelerators
  - Standard in high-end SoCs, domain-specific hard blocks in FPGAs.
- Many Accelerator IPs are configurable:
  - Optional extensions, tile count, memory hierarchy, etc.

Hardware is getting harder

- How can developers write Software for this in the first place?
- How can developers afford to build generation-specific Software?

Next-Gen compilers are needed!

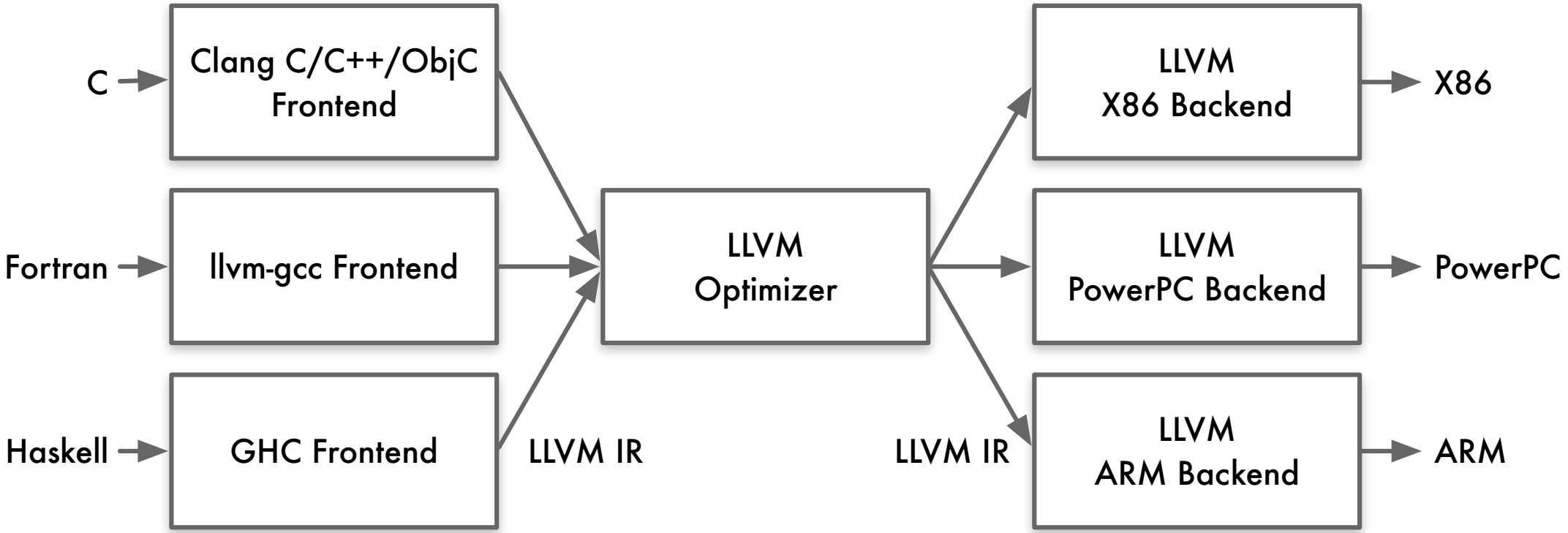
What we need:

- Hardware abstraction spanning diverse accelerators.
- Support for heterogeneous compute platforms.
- Domain specific languages and programming models.
- Quality, reliability and scalability of infrastructure.

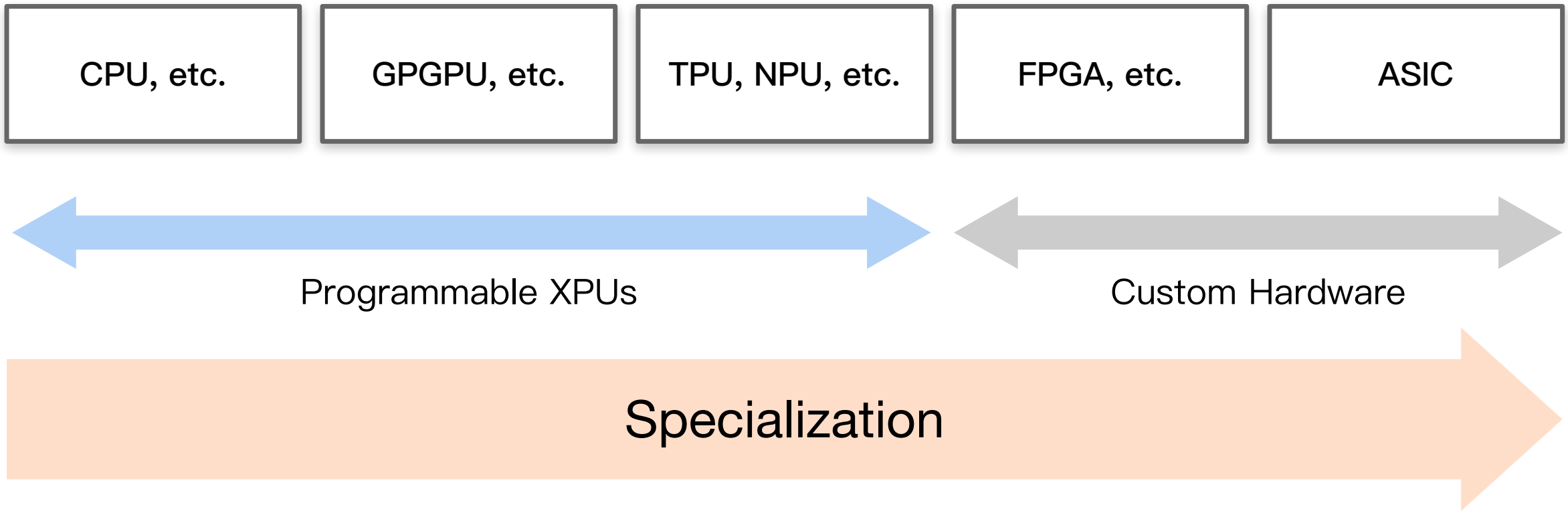
C Compilers leading into the early 90s

Three Phase Compiler Stage

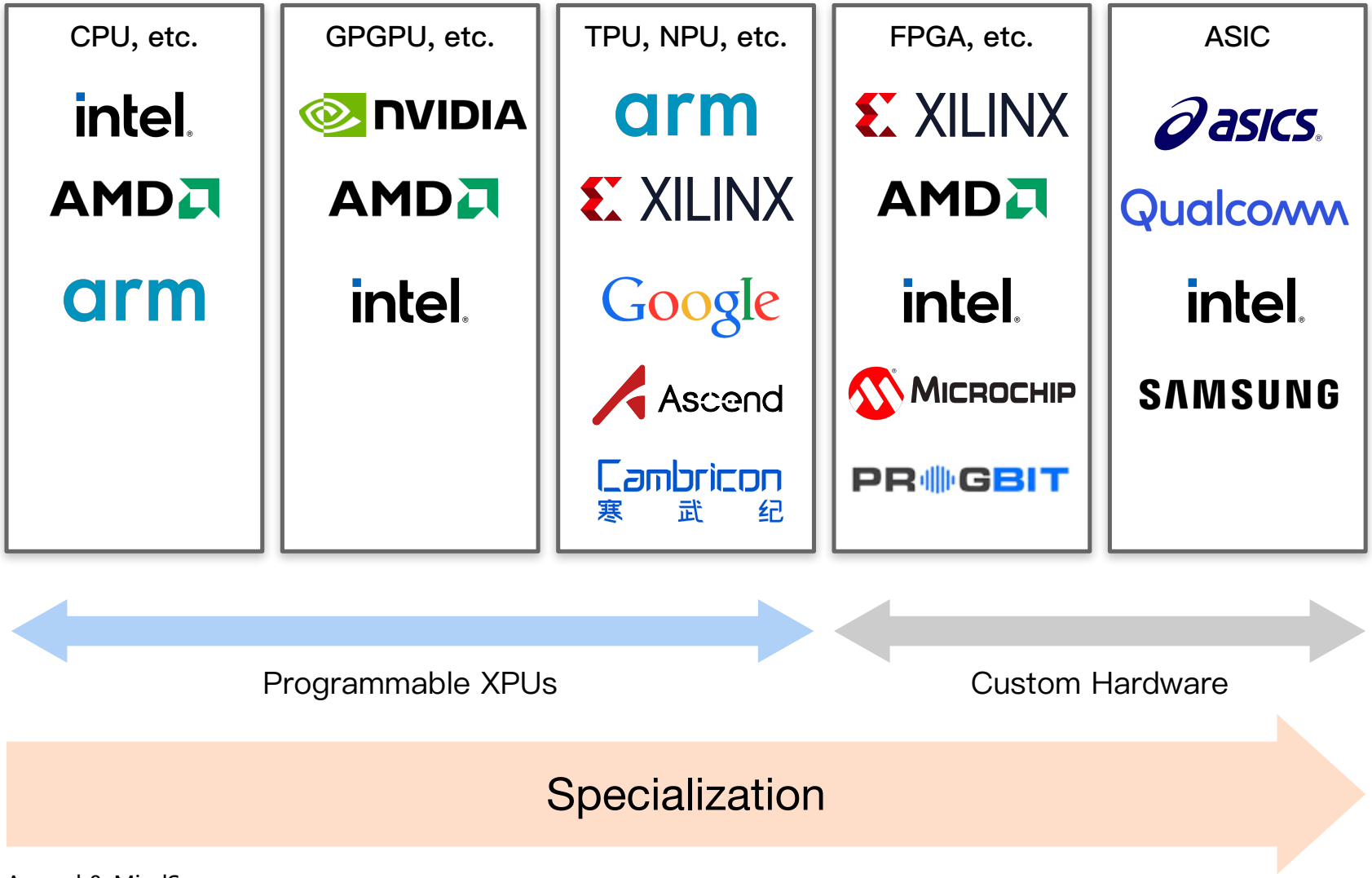
LLVM Enables Collaboration & Reuse



It's happening!



Los of players!



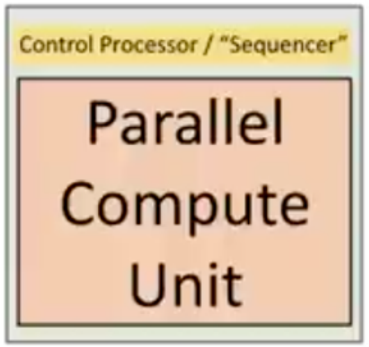
How do we compile for this?

cādence

⇒ Not very compatible, inconsistent quality and scope

- … and don’t share much code

How do accelerators work?



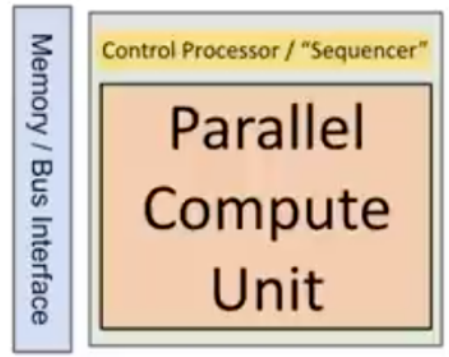
Control Processor / Sequencer

- Executes commands by the host driver app
- Handles booting and other housekeeping
- Diagnostics, security, debug, other functions

Some accelerators may do significantly more!

Ratio of control to parallel compute vary, as do the internal arch's of both

Add a system interface

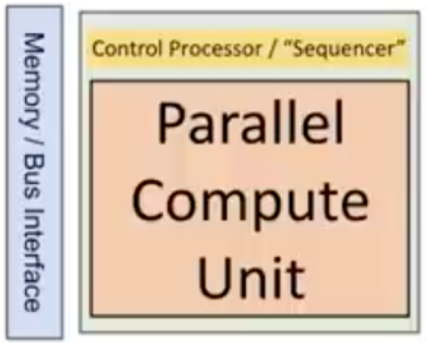


Communicate w other parts of the SoC, or to off-chip resources

Including DDR, HBM, … AMBA, PCI, CXL, etc depending on integration level

“Oops we need some software”

Hardware



Software

Programming Model + Userspace API

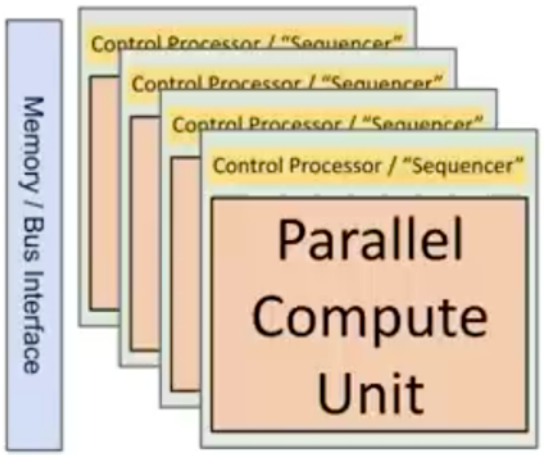
Device Kernels

Control Proc Assembler + Kernel Driver

The SW people are called in after the accelerator is defined to “make it work”

Larger accelerators go multicore/SIMT…

Hardware



Software

Programming Model + Userspace API

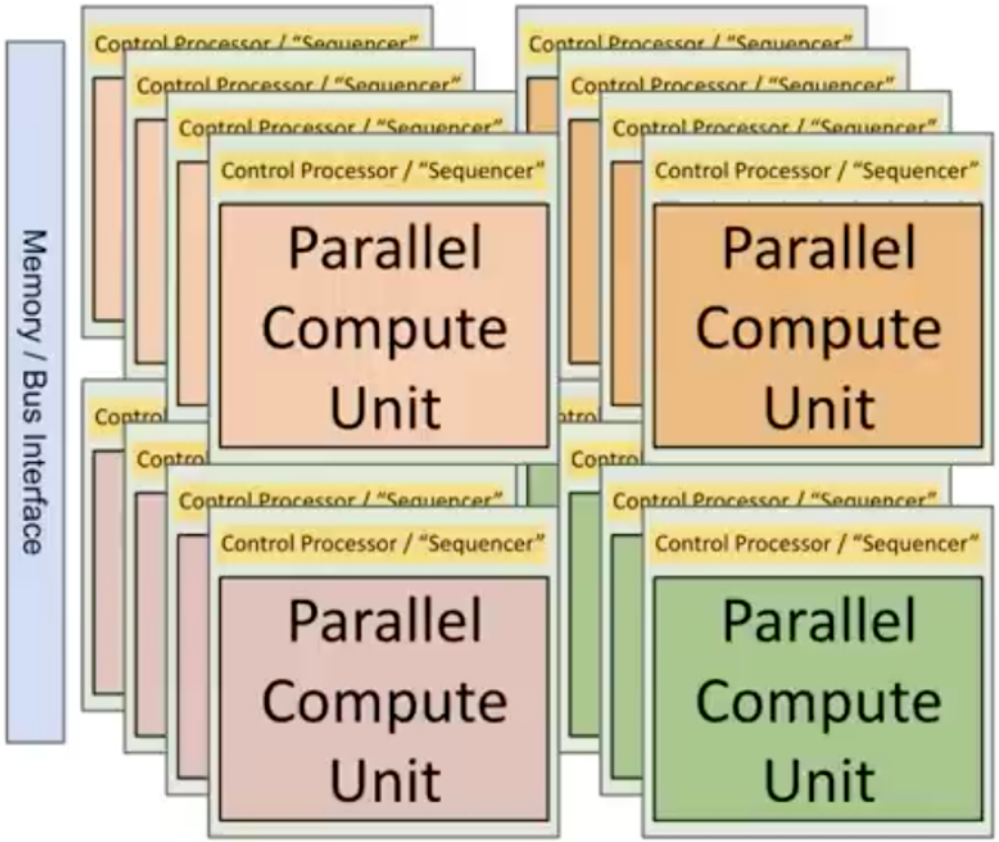
Parallel Device Kernels

Control Proc Assembler + Kernel Driver

Use of more HW area is desired, requiring parallel control logic

Tiling and heterogeneity for generality

Hardware



Software

Programming Model + Userspace API

Multistream Mgmt / Interop Parallelism
Memory + Communication Optimization
Heterogenous Device + Host fallback

Parallel Device Kernels

Control Proc Assembler + Kernel Driver

⇒ Also, hierarchical compute at the board, rack, and datacenter level

Pro & Cons of hand written kernels

Benefits:

- Easy to get started, ability to get peak performance, hackability

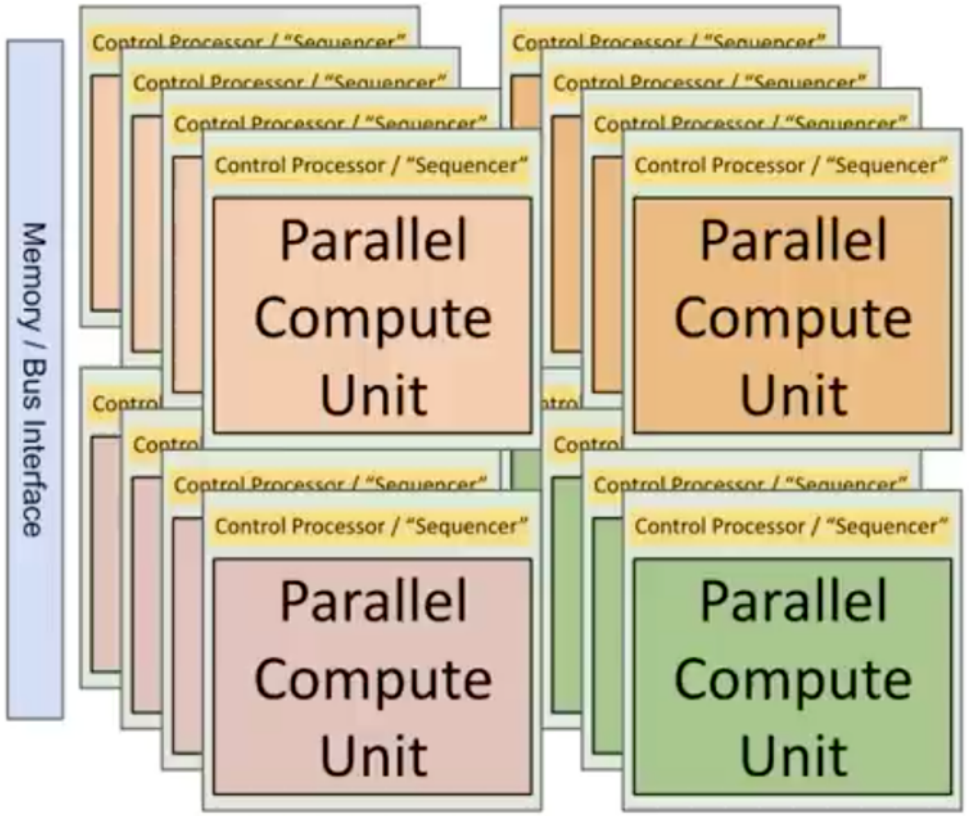
Problem: **hand written kernels don’t scale**

- Expensive to maintain a library of 100’s to 1000’s of kernels
- Don’t scale to configurable IPs, not even memory hierarchy dimensions
- Don’t scale to device families, or evolving µarch’s over time
- Eventually end up limiting HW design space exploration / evolution

Often addressed with metaprogramming (aka “mini compilers”)

“DSA Compilers” to the rescue

Hardware



Software

Programming Model + Userspace API

Accelerator Kernel Compiler

Multistream Mgmt / Interop Parallelism
Memory + Communication Optimization
Heterogenous Device + Host fallback
Kernel Code Generation

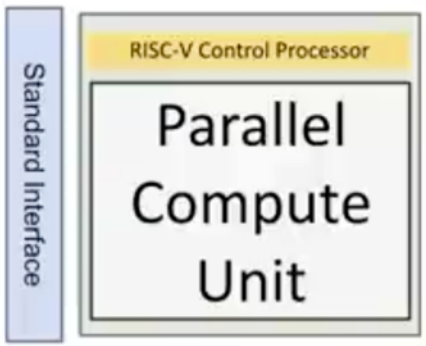
Control Proc Assembler + Kernel Driver

This is Hard!

… And we keep reinventing it over and over again.

… at the expense of usability and quality.

Standardize your base Software



Write your kernels in C or LLVM IR!

- Use *existing* code generators
- Use *existing* simulators
- Step through them in a **debugger**

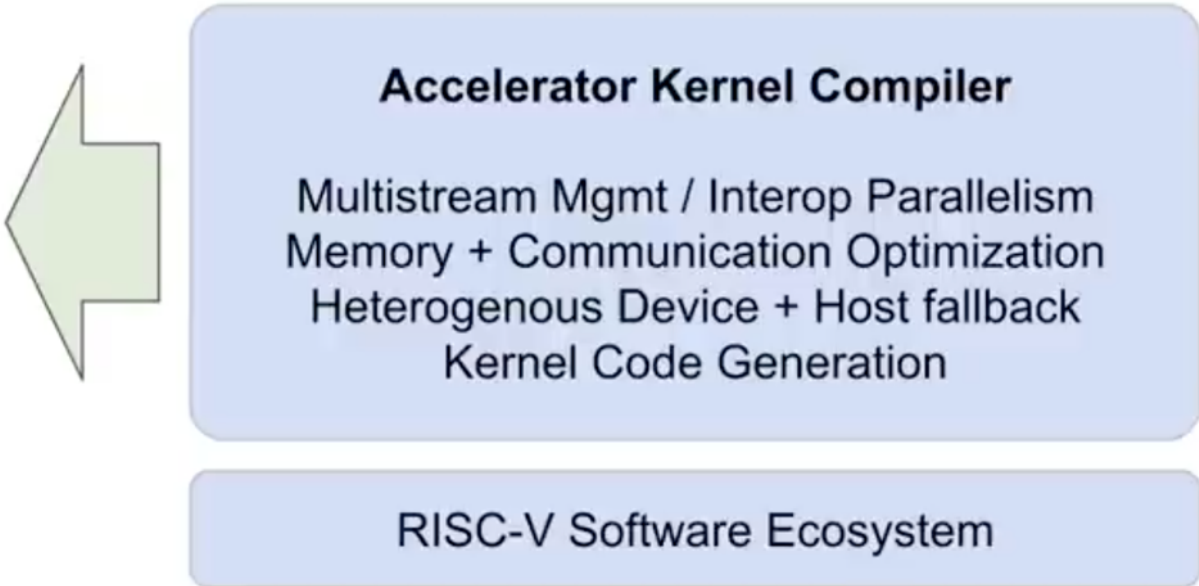
RISC-V **Compiler** + Kernel Drivers

The next frontier: DSA Compilers?

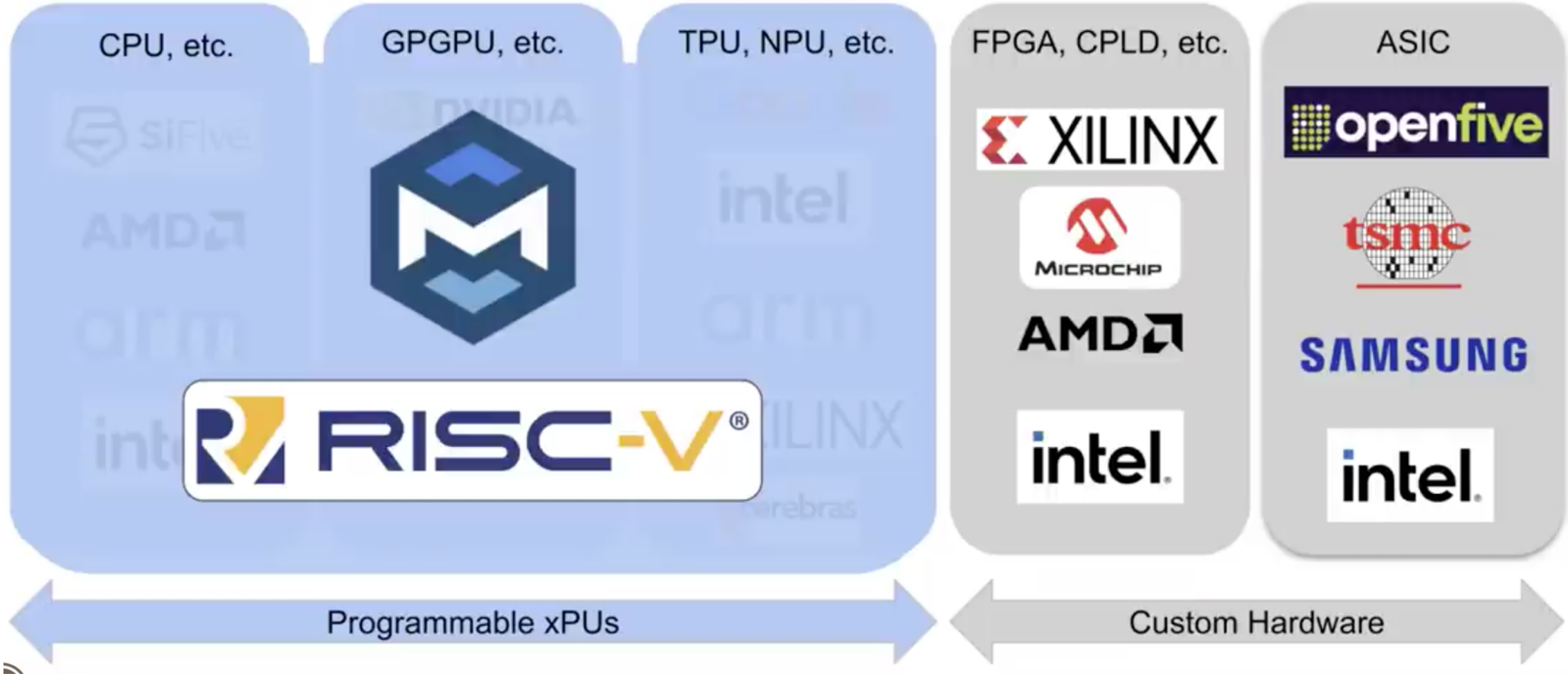
"No one size fits all" compiler!

Shape of the problem is the same...
... but the accel details always vary

How do we get reuse?

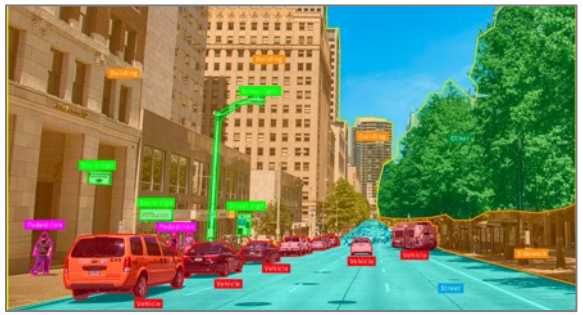


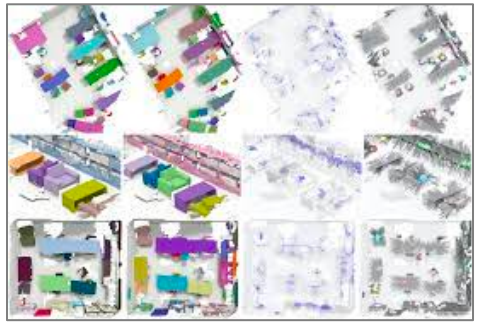
RISC-V+MLIR: Uniting an Industry



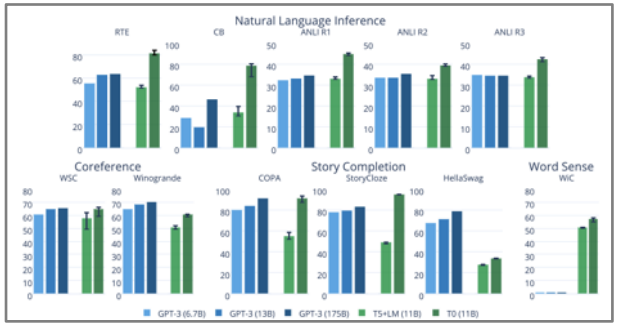
Deep Learning is widely used

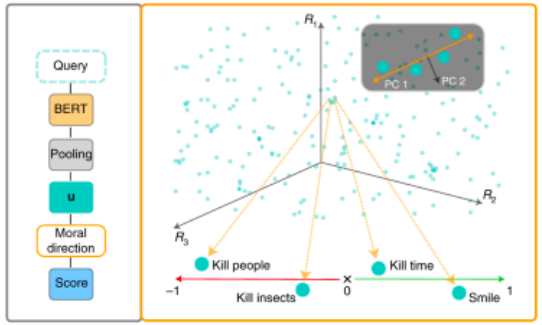
CV

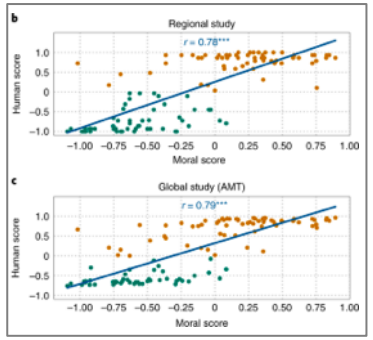




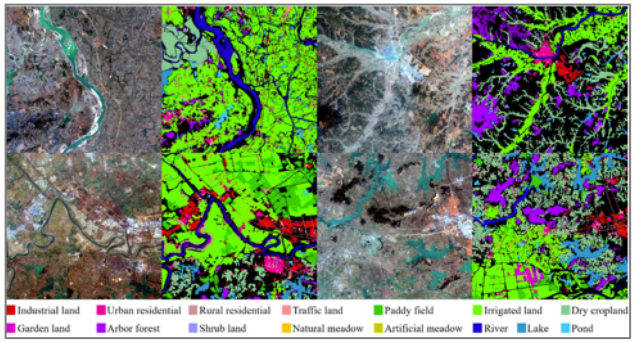
NLP

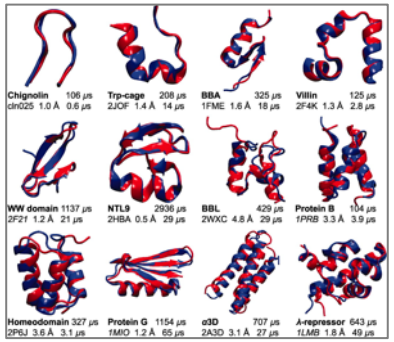


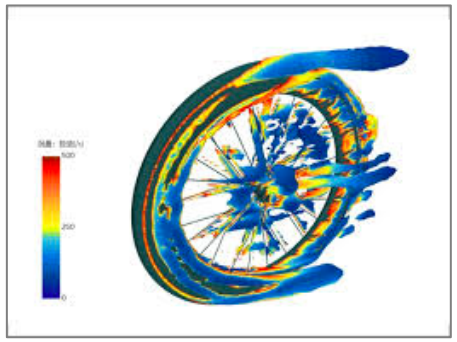


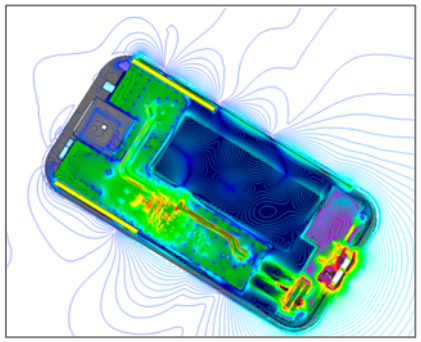


Science

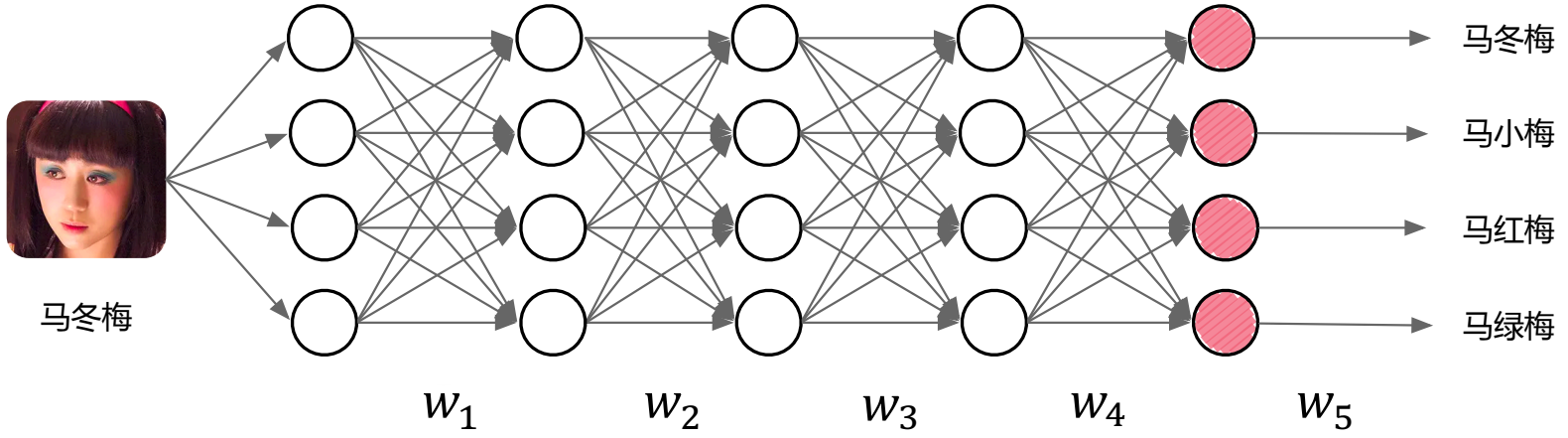






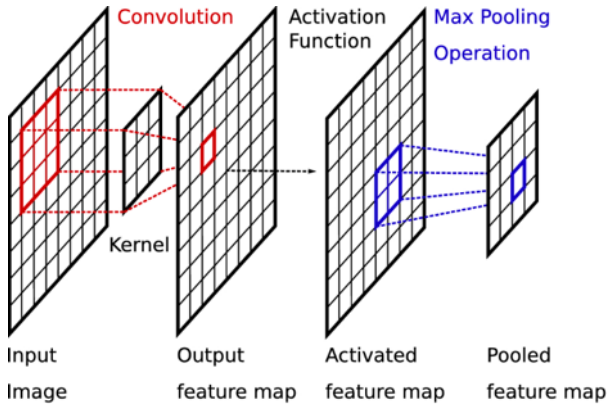


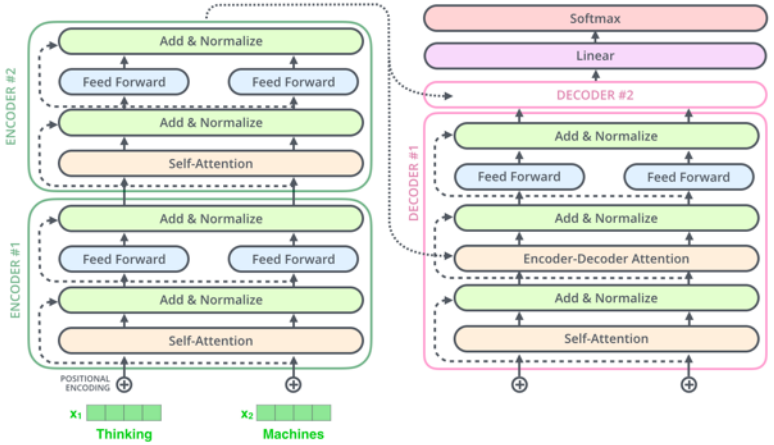
AI Framework Problem

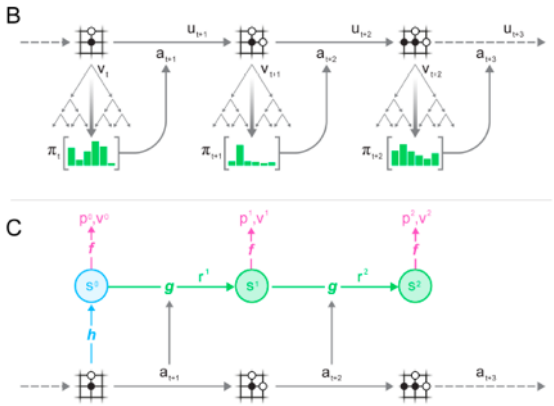


MindSpore

AI Framework Problem



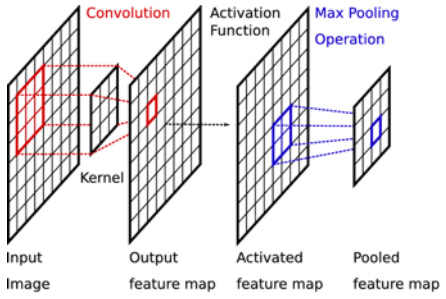


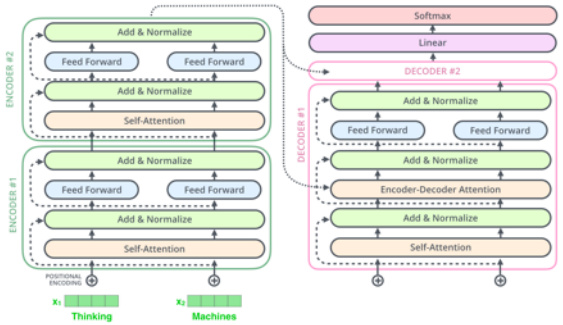


[M]S 昇思 MindSpore

www.hiascend.com
www.mindspore.cn

The Gap Between SW and HW is huge!





昇思 MindSpore

TensorFlow

ONEFLOW

飞桨

PyTorch

www.hiascend.com
www.mindspore.cn

C Compilers leading into the early 90s

Challenge

1. 越来越多新算子被提出，算子库的开发、维护、优化和测试工作量指数上升；
2. 专用加速芯片爆发导致性能可移植性成为一种刚需；

Challenge: Operator

- 越来越多新算子被提出，算子库的开发、维护、优化和测试工作量指数上升：
  - 增加新算子，硬件不仅需要实现，还需要结合硬件进行特性优化和测试，尽量充分发挥硬件性能。以 Convolution 运算为例，需要将 Convolution 操作转换为 GEMM 矩阵乘法；算法新提出新的 Swish 算子，硬件需要新增 Swish 对应实现。
  - 硬件供应商还会有针对性发布优化库（如 MKL-DNN 和 CuDNN）。但是对于专用硬件，需要提供开发类似的优化库，不仅会增加大量算子优化、封装的工作，还会过于依赖库无法有效利用专用硬件芯片能力。

Challenge: Optimization

- 专用加速芯片爆发导致性能可移植性成为一种刚需：
  - 大多数 NPU 使用 ASIC，在神经网络场景对计算、存储和数据搬运 做了特殊的指令优化，使得对 AI 相关的计算会提升性能，如 NVIDIA 虽然作为 GPGPU，但是 DGX 系列提供专用的 Tensor Code。
  - 不同厂商提供 XPU 的 ISA（Instruction Set Architecture）千奇百怪，一般缺乏如 GCC、LLVM 等编译工具链，使得针对 CPU 和 GPU已有的优化算子库和针对语言的优化 Pass 很难短期移植到 NPU 上。

无限的算力和有限的精力

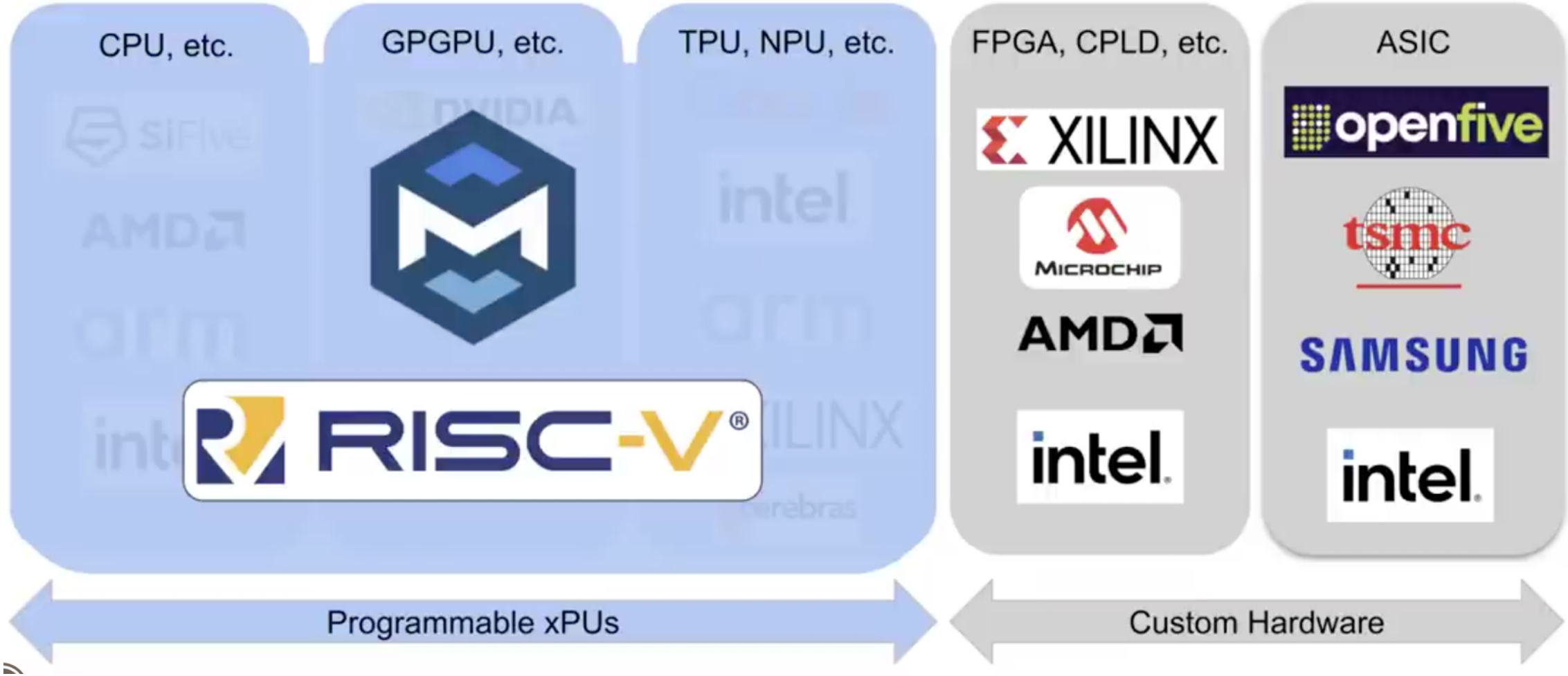
古有精卫填海

我想用钱试试

18岁的吴彦祖

你说我像苏大强？

RISC-V+MLIR: Uniting an Industry



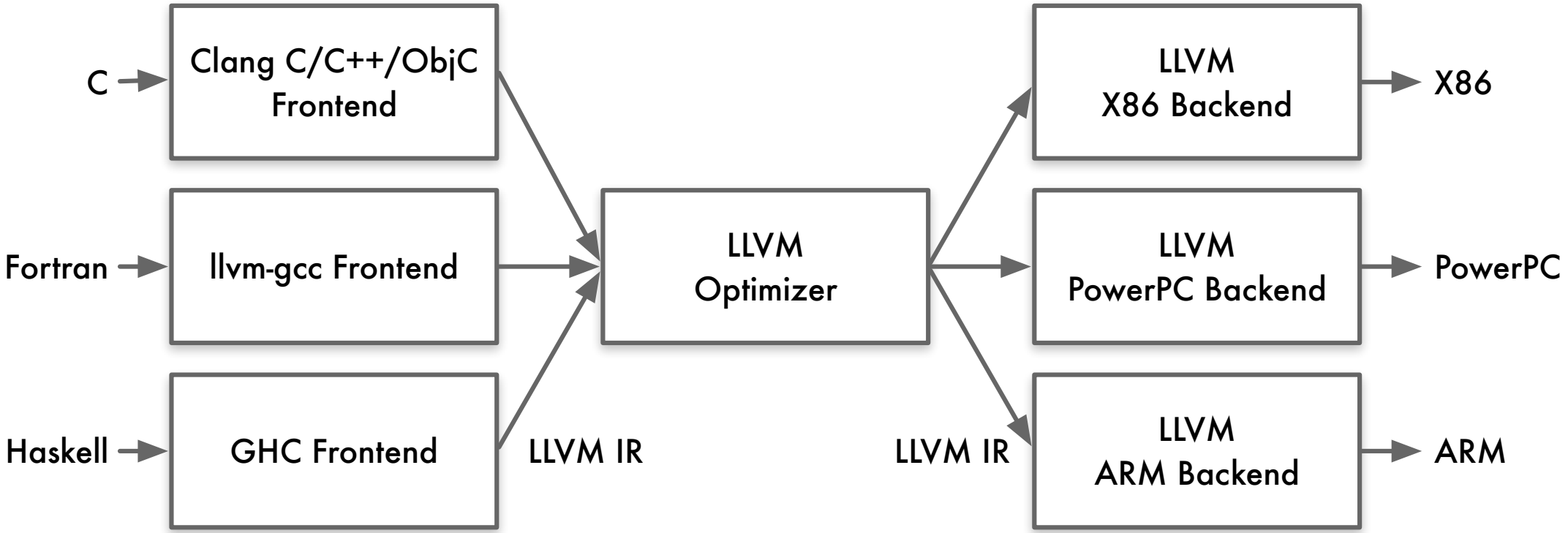
传统编译器与AI编译器

AI 编译器跟传统编译器有什么区别吗？

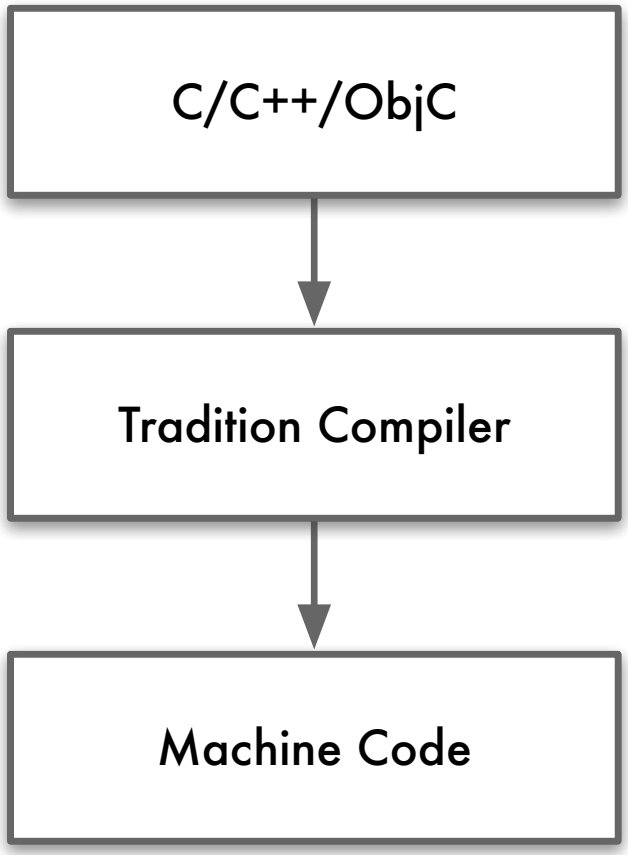
Likewise

- **目标相同：**通过自动化方式进行程序优化和代码生成，从而降低对不同硬件的手工优化；
- **优化方式类似：**在编译优化层通过统一 IR 执行不同的Pass进行优化，从而提高执行性能；
- **软件结构栈类似：**分成前端、优化、后端三段式，IR 解耦前端和后端使得模块化表示；
- **AI编译器依赖传统编译器：**AI编译器对 Graph IR 进行优化后，将优化后的 IR 转换成传统编译器 IR，最后依赖传统编译器针进行机器码生成。

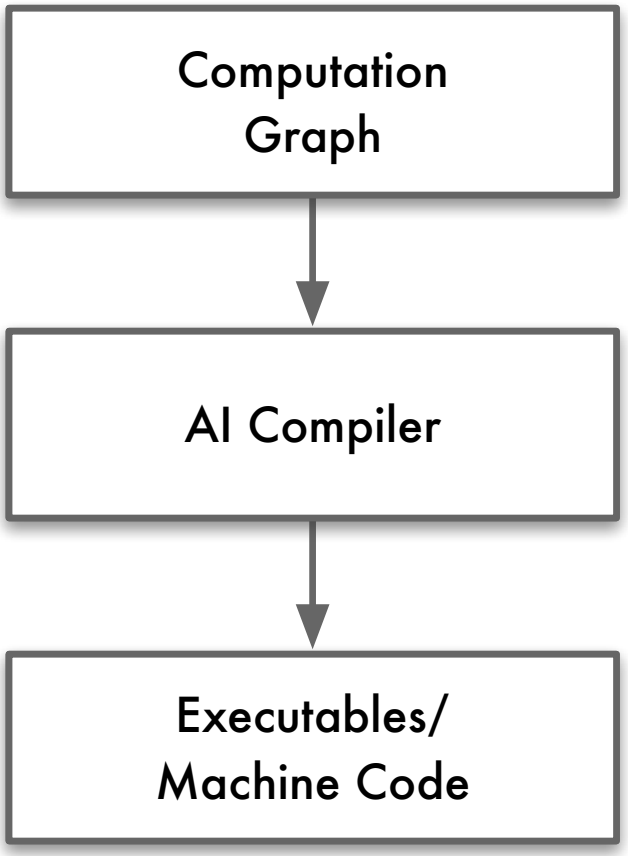
Difference



Difference

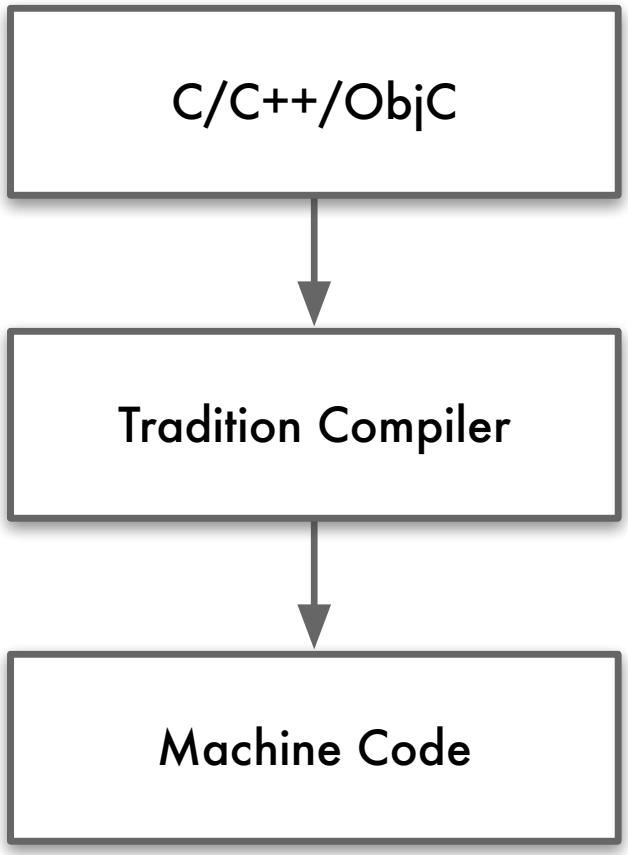


输入高级语言输出低级语言



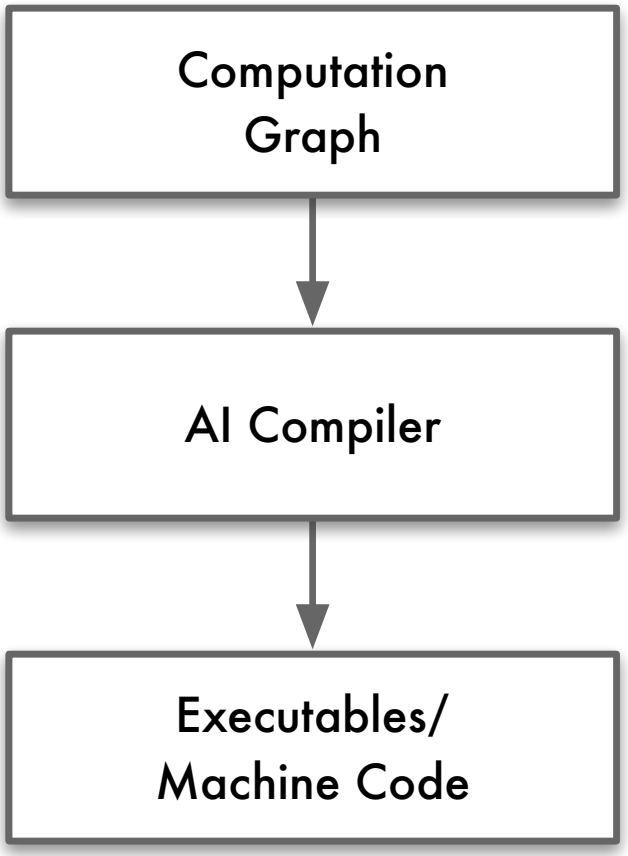
输入计算图/算子，输出低级语言

Difference



主要问题是降低编程难度

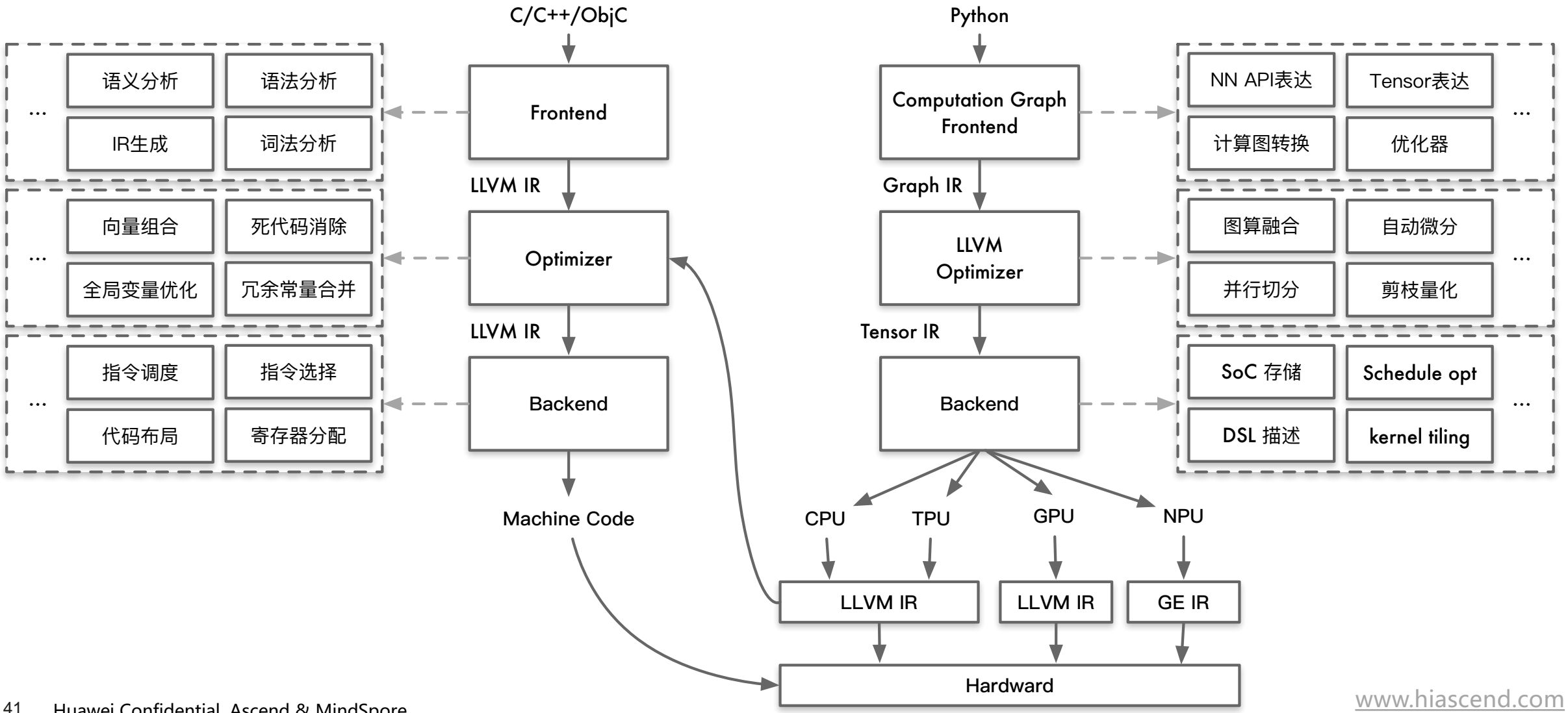
其次是优化程序性能



主要优化程序性能

其次是降低编程难度

Difference



Difference

- **IR 差异：**AI 编译器的 IR 与传统编译器的IR所抽象出来的概念和意义并不相同。
  - AI编译器一般会有 high-level IR，用来抽象描述深度学习模型中的运算，如：Convolution、Matmul 等，甚至部分会有 Transformer 带有图的结构。
  - 传统编译器相对而言 low-level IR，用于描述基本指令运算，如 load、store 等。有了high-level IR，AI编译器在描述深度学习模型类 DSL 更加方便。

Difference

- **优化策略：**AI 编译器面向AI领域，优化时引入更多领域特定知识，从而进行更 high-level，更加 aggressive 优化手段。如：
  - AI编译器在 high-level IR 执行算子融合，传统编译器执行类似 loop fusion 时候，往往更加保守。缺点是可能会导致调试执行信息跟踪难；
  - AI编译器可以降低计算精度，比如int8、fp16、bf16等，因为深度学习对计算精度不那么敏感。但传统编译器一般不执行改变变量类型和精度等优化。