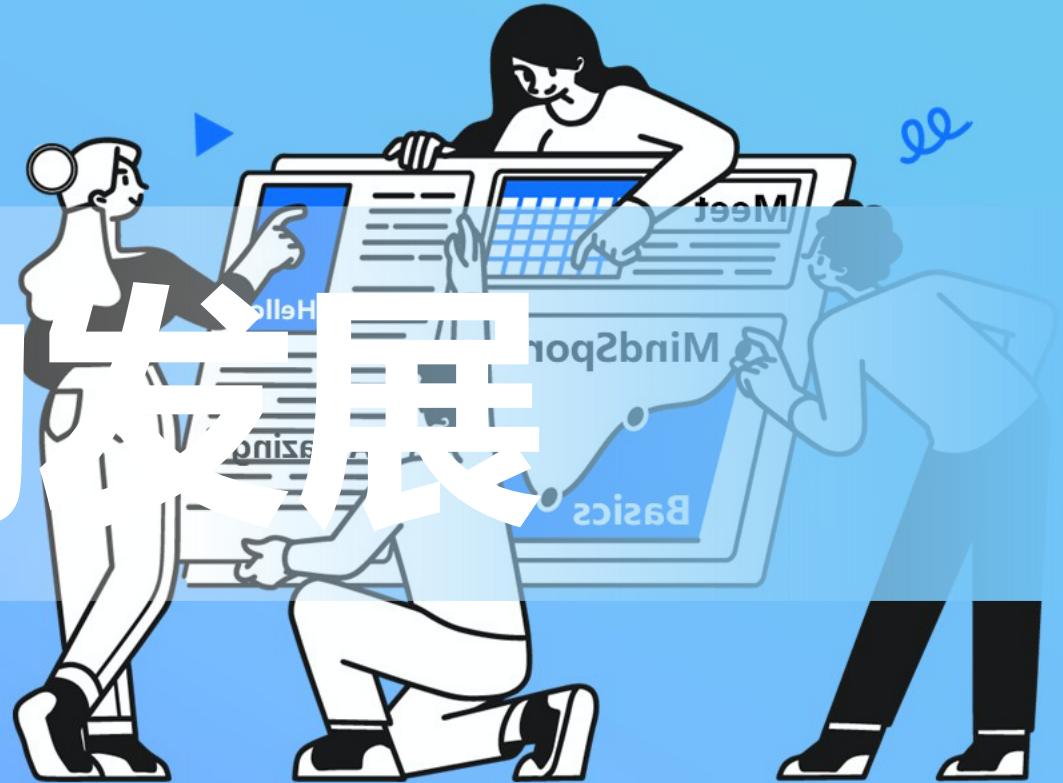


AI编译器系列

AI 编译器的进展



ZOMI



BUILDING A BETTER CONNECTED WORLD

Ascend & MindSpore

www.hiascend.com
www.mindspore.cn

Talk Overview

I. 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

AI 编译器对比

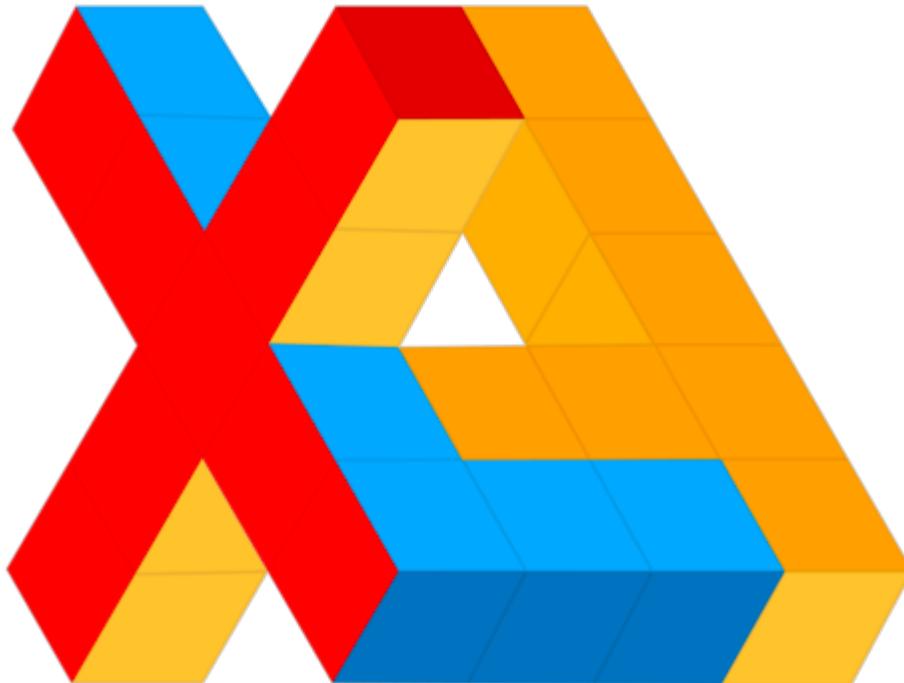
The Deep Learning Compiler: A Comprehensive Survey

Table 1. The comparison of DL compilers, including TVM, nGraph, TC, Glow, and XLA.

		TVM	nGraph	TC	Glow	XLA
	Developer	Apache	Intel	Facebook	Facebook	Google
Frontend	Programming	Python/C++ Lambda expression	Python/C++ Tensor expression	Python/C++ Einstein notation	Python/C++ Layer programming	Python/C++ Tensorflow interface
	ONNX support	✓ tvm.relay.frontend .from_onnx (built-in)	✓ Use ngraph-onnx (Python package)	✗	✓ ONNXModelLoader (built-in)	✓ Use tensorflow-onnx (Python package)
	Framework support	tvm.relay.frontend .from_* (built-in) tensorflow/tflite/keras pytorch/caffe2 mxnet/coreml/darknet	tensorflow paddlepaddle (Use *-bridge, act as the backend)	(Define and optimize a TC kernel, which is finally called by other frameworks.) pytorch/other DLPack supported frameworks	pytorch/caffe2 tensorflowlite (Use built-in ONNXIFI interface)	Use tensorflow interface
	Training support	✗ Under developing (Support derivative operators now)	✓ Only on NNP-T processor	✓ (Support auto differentiation)	✓ (Limited support)	✓ Use tensorflow interface

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。



XLA：优化机器学习编译器

- XLA：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、小算子如何重新融合、新的大算子如何生成，整体设计主要通过HLO/LLO/LLVM IR 实现，所有 Pass 规则都是手工提前指定。



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。



TVM：端到端深度学习编译器

- **TVM**：分为Relay和TVM两层，Relay关注图层，TVM关注算子层，拿到前端子图进行优化， Relay关注算子间融合、TVM关注新算子和kernel生成，区别在于TVM开放架构，Relay目标是可以接入各种前端，TVM也是一个可以独立使用的算子开发和编译的工具，算子实现方面采用 Compute（设计计算逻辑）和 Schedule（指定调度优化逻辑）分离方案。



TENSOR COMPREHENSIONS : 神经网络语言编译器

- TensorComprehensions 是一种可以构建 just in time(JIT)系统的语言,程序员可通过该语言用高级编程语言去高效的实现 GPU 等底层代码。



TENSOR COMPREHENSIONS : 神经网络语言编译器

- **TC**：算子计算逻辑的较容易实现，但 Schedule 开发难，既要了解算法逻辑又要熟悉硬件体系架构，此外图算边界打开后，小算子融合后，会生成新算子和 kernel，新算子 Schedule 很难生成，传统方法定义 Schedule 模板；TC 希望通过 Polyhedral model 实现 auto schedule。



nGraph : 兼容所有框架的深度学习系统编译器

- nGraph 的运作应作为深度学习框架当中更为复杂的 DNN 运算的基石，并确保在推理计算与训练及推理计算之间取得理想的效率平衡点。



AI 编译器的输入：计算图



- 

1 小时搞清楚 Data Flow Graph Computation graph
计算图系列 01. 内容介绍
01:55

【计算图】系列第一篇！计算图有哪些内容知识？

291 10-6
- 

2 小时搞清楚 Data Flow Graph Computation graph
计算图系列 02. 什么是计算图
10:31

【计算图】系列第二篇！为什么AI框架都用计算图？什么是

319 10-8
- 

3 小时搞清楚 Data Flow Graph Computation graph
计算图系列 03. 跟自动微分什么关系
16:45

【计算图】系列第三篇！计算图跟微分什么关系？怎么用计

280 10-9
- 

4 小时搞清楚 Data Flow Graph Computation graph
计算图系列 04. 图优化与执行调度
10:30

【计算图】第四篇！图优化与执行调度！计算图优化！单算

248 10-10
- 

5 小时搞清楚 Data Flow Graph Computation graph
计算图系列 05. 怎么表示控制流
19:27

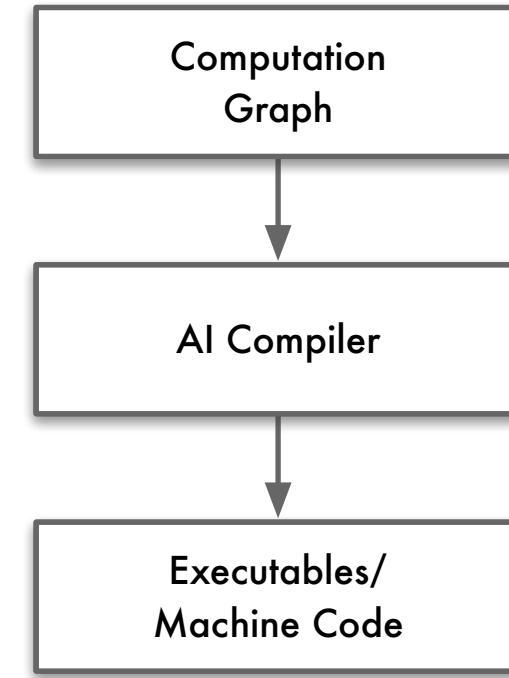
【计算图】第五篇！PyTorch 控制流如何实现？动静统一原

294 10-11
- 

6 小时搞清楚 Data Flow Graph Computation graph
计算图系列 06. 挑战和未来
06:42

【计算图】第六篇！计算图未来将会走向何方？

202 10-12



计算图的产生：AI框架



【AI框架基础】系列第一篇！
介绍分享内容！AI框架基础、

222 10-4



【AI框架基础】系列第二篇！
AI框架有什么用？没有AI框架

205 10-5



【AI框架基础】系列第三篇！
AI框架之争！都2022年，还在

190 10-5



【AI框架基础】系列第四篇！
函数式编程和声明式编程有什

144 10-6

AI 编译器的挑战

Challenge

1. 动态 Shape问题
2. Python 编译静态化
3. 发挥硬件性能，特别是DSA类芯片
4. 处理神经网络特性：自动微分、自动并行等
5. 易用性与性能兼顾

Challenge I: Dynamic shape

动态 shape 和动态计算图：

- 现阶段主流的AI编译器主要针对特定静态 shape 输入完成编译优化，对包含 Control Flow 语义的动态计算图只能提供有限的支持或者完全不能够支持。
- AI 的应用场景却恰恰存在大量这一类的任务需求。AI编译器前端将计算图改写为静态计算图，或者将适合编译器部分子图展开交给编译器进行优化。
- 部分 AI 任务并不能通过人工改写来静态化（如金字塔结构的检测模型），导致这些情况下编译器完全难以有效优化。

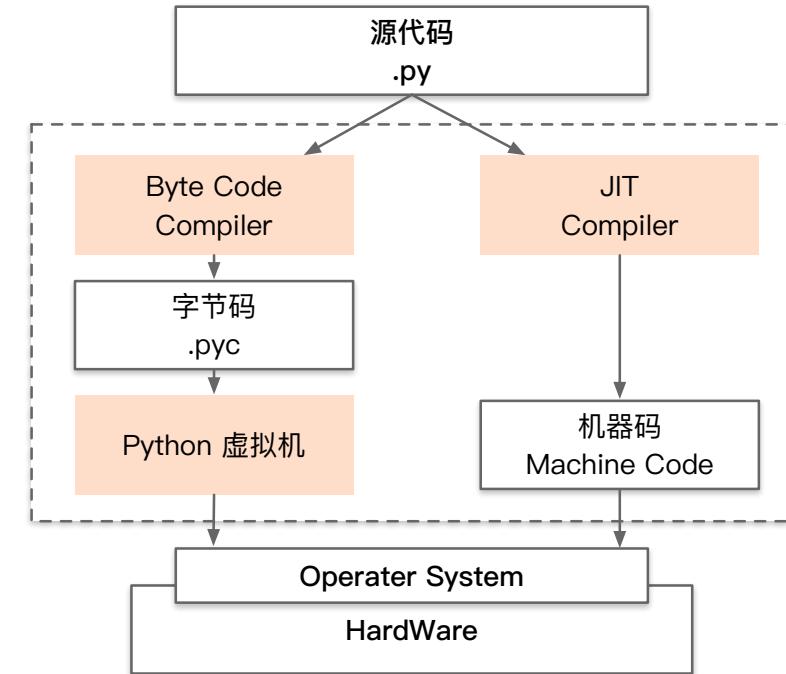
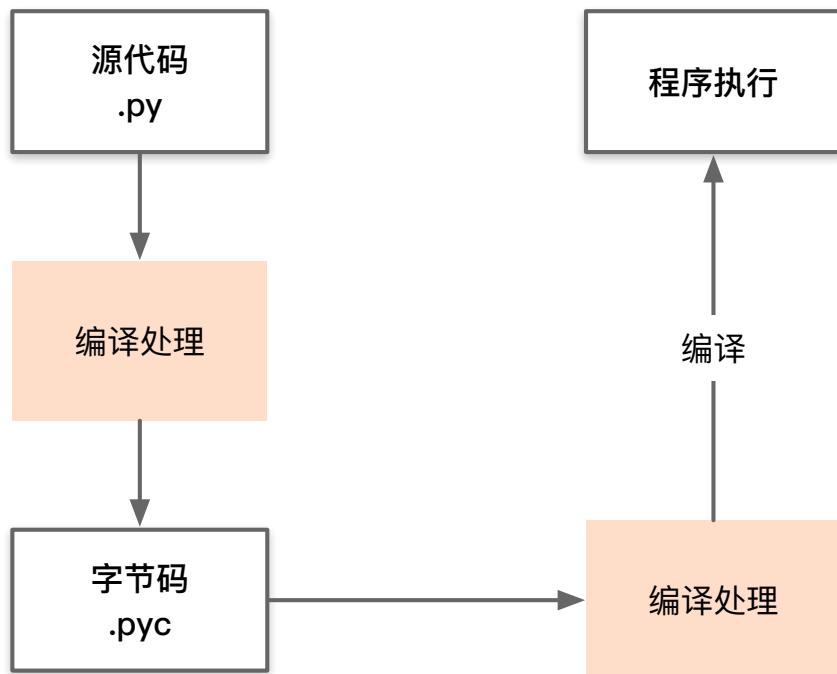
Challenge I: Python Static 静态化

通过 JIT，Python 程序执行静态编译优化，从而提升性能、方便部署。业界 JIT 通用做法：

1. **Python JIT 虚拟机**：期望在 Python 解释执行的基础上增加 JIT 编译加速的能力，典型的如 PyPy 或者 CPython，但由于前期设计问题 Python JIT 虚拟机兼容难。
2. **修饰符方式**：Python JIT 通过修饰符，进行部分 Python 语句加速。但无论是 `torch.jit.script`、`torch.jit.trace`、`torch.fx`、`LazyTensor`、`TorchDynamo`，几乎把能用的手段都用上，还是缺乏一个完备的方案。

Challenge I: Python Static 静态化

Python 执行时，首先会将.py文件中的源代码编译成Python的byte code(字节码)，然后再由Python Virtual Machine 来执行这些编译好的 byte code。实际上，Python Virtual Machine 是一种抽象层次更高 Virtual Machine。基于C的 Python 编译出的字节码文件，通常是.pyc格式。

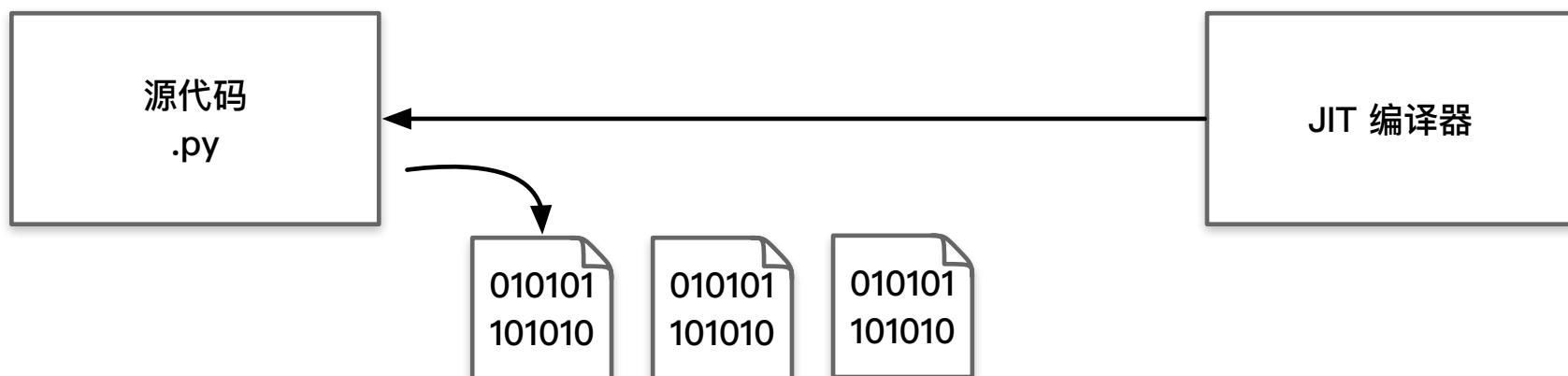


Challenge I: Python Static 静态化

- I. **CPython** : CPython 混合了编译和解释功能，将Python源代码首先被编译成一系列中间字节码，然后由CPython虚拟机内部 while 循环不断匹配字节码并执行对应字节码指令 case 分支内部的多条C函数。



- I. **PyPy** : 基于RPython语言构建，利用 JIT 即时编译来执行 Python 代码。不同于解释器，并不会逐行运行代码，而是在执行程序前先将部分代码编译成机器码。



Challenge I: Python Static 静态化

AI框架静态化的方案普遍采用修饰符这套方案，这套方案细分下来也有两种不同的方法：

- Tracing Based , 如 PyTorch.fx
- AST Transform , 如 PyTorch.jit.trace

```
@torch.fx.wrap
def torch_randn(x, shape):
    return torch.randn(shape)

def f(x):
    return x + torch.randn(x, 5)
fx.symbolic_trace(f)
```

```
def foo(x, y):
    return 2 * x + y

traced_foo = torch.jit.trace(foo, (torch.rand(3), torch.rand(3)))

@torch.jit.script
def bar(x):
    return traced_foo(x, x)
```

Challenge I: Python Static 静态化

AI编译器在 Python 静态化方面的挑战：

- 类型推导：从 Python 动态类型到编译器 IR 静态类型
- 控制流的表达：if、else、while、for 等控制表达
- 灵活的语和数据类型转换：针对 Slice、Dict、Index 等操作
- JIT的编译性能：Tracing Based 还是 AST Transform 都需要增加额外的编译开销

Challenge II: Toward DSA

DSA 专用硬件架构：

- AI 训练和推理都是对性能敏感，所以在AI的场景中大量用到加速器，包括CPU SIMD单元、GPU SIMT架构、华为昇腾Ascend Cube核等专用架构，NV GPU H100架构DSA特征也逐步明显。
- 因此 AI 编译器逐步成为发挥多样性算力的关键，特别是近期 Dataflow + SIMD 类 DSA芯片占比逐步提升的情况下。

Dataflow：

- Dataflow 的执行调度更加能发挥芯片的性能，即芯片进行整图或者子图的调度，而不是像GPU那样，主流是 kernel by kernel的调度。

Challenge II: Toward DSA

编译器在性能优化的难度和复杂度挑战变大：

- 性能优化依赖图算融合优化，图层和算子层独立优化无法充分发挥芯片性能，需要图算融合优化，例如子图切分、子图内垂直融合优化和水平并行优化；
- 优化复杂度提升，标量+向量+张量+加速指令、多级的存储结构，导致 Kernel 实现 Schedule、Tiling、Vectorization、Tensorization 复杂；

Challenge II: Toward DSA

当前 AI 编译器没有形成统一完善的方案：

- 打开图和算子的边界，进行重新组合优化，充分发挥芯片性能；
- 多种优化手段融合：垂直融合优化（ Buffer Fusion等）和水平并行（ Data Parallel等）优化；
- 重新组合优化后的子图的Kernel代码自动生成，如scheduling、 tiling、 vectorizing；

Challenge III: Specific Optimizations

自动并行，当前大模型训练遇到内存墙、性能墙等依赖复杂的并行策略来解决：

- Scale out：多维混合并行能力，数据并行、张量并行、流水线并行等
- Scale up：重计算、混合精度、异构并行等

Scale Out/Up 最大挑战就是效率墙：

- 依赖手工配置切分策略，对开发者来说，门槛高，效率低；
- 半自动并行解决一部分效率，真正要解放依赖编译+凸优化问题，自动找到最优并行策略。

Challenge III: Specific Optimizations

面向 HPC 场景自动微分的要求更高：

- 控制流：动态图通过Python侧执行控制流，一旦循环次数多的话，性能劣化；静态图的自动微分通过逻辑拼接或者计算图展开方式来解决，能解决一定性能问题但是方案有待完善。
- 高阶微分：高阶的前向微分和后向微分如何通过 Jacobian matrix 有效展开，高阶微分通过 Hessian matrix 如何模拟快速计算，开发者如何灵活对高阶微分的图进行控制？

Challenge III: 易用性与性能兼顾

与AI框架的边界和对接：

- 不同AI框架对深度学习任务的抽象描述和 API 接口不同，语义和机制上有各自的特点。需要考虑如何在不保证所有算子被完整支持的情况下透明化的支持用户的计算图描述。

Challenge III: 易用性与性能兼顾

对用户透明性问题：

- 部分 AI 编译器并非完全自动的编译工具，性能表现依赖于用户提供的高层抽象的实现模版，如TVM。主要是为算子开发工程师提供效率工具，降低用户人工调优各种算子实现的人力成本。
- 现有抽象却常常无法足够描述创新的硬件体系结构上所需要的算子实现。需要对编译器架构足够熟悉的情况下对其进行二次开发甚至架构上的重构，门槛及开发负担仍然很高。

Challenge III:易用性与性能兼顾

编译开销：

- AI 编译器作为性能优化工具，只有在编译开销对比带来的性能收益有足够优势才有实用价值。部分应用场景下对于编译开销的要求较高。对于开发者而言，使用AI编译器阻碍其快速的完成模型的调试和验证工作，极大增加开发和部署的难度和负担。

Challenge III:易用性与性能兼顾

性能问题：

- 编译器的优化本质上是将人工的优化方法，或者人力不易探究到的优化方法通过泛化性的沉淀和抽象，以有限的编译开销来替代手工优化的人力成本。深度学习编译器只有在性能上真正能够代替或者超过人工优化才能真正发挥价值。

Challenge III:易用性与性能兼顾

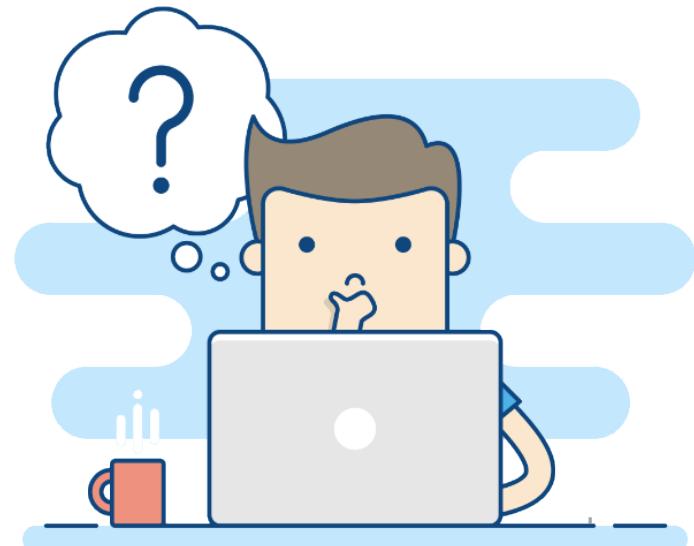
鲁棒性：

- AI 编译器大多数处于研究型，产品成熟度距离工业级应用有较大差距。是否能够顺利对计算图编译，计算结果的正确性，对错误进行跟踪调试都需要考虑。

Question?

- 图算能否统一表达，统一编译优化，形成通用的 AI 编译器？

当前的AI框架下，图层和算子层是分开表达和优化，算法工程师主要是接触图层表达，AI框架、芯片、kernel开发工程师主要是负责算子的表达，未来能否会有IR打破图算之间的GAP，未来在AI+科学计算、AI+大数据等新场景驱动下，使得计算图层和算子层不再清晰，能否统一AI的编译优化？



Question?

- 完全的自动并行是否可行？

自动并行能根据用户输入的串行网络模型和提供的集群资源信息自动进行分布式训练，通过采用统一分布式计算图和统一资源图设计可支持任意并行策略和各类硬件集群资源上分布式训练，并且还能利用基于全局代价模型的规划器来自适应为训练任务选择硬件感知的并行策略。实际上自动并行是一个策略搜索问题，策略搜索能够在有限的搜索空间找到一个次有的答案，但是真正意义上的自动并行能否做到需要进一步思考和验证。



Question?

- AI芯片需要编译器吗？AI芯片需要AI编译器吗？

AI芯片对于编译器的依赖取决于芯片本身的设计。越灵活的芯片对于编译器的依赖会越大。在AI芯片设计之初，有CISC风格把优化在芯片内部解决。但是随着专用领域的演化，为了支持更加灵活的需求，AI芯片本身会在保留张量指令集和特殊内存结构的前提下越来越灵活。未来的架构师需要芯片和系统协同设计，自动化也会越来越多地被应用到专用芯片中去。



AI 编译器的未来

Future

1. 编译器形态：分开推理和训练，AOT 和 JIT 两种编译方式并存
2. IR 形态：需要有类似于 MLIR 对 AI 统一的 IR 表达
3. 自动并行：提供跨机器、跨节点自动并行的编译优化能力
4. 自动微分：提供高阶微分的计算方式，并方便对图进行操作
5. kernel自动生成：降低开发门槛，快速实现高效和高泛化的算子



BUILDING A BETTER CONNECTED WORLD

THANK YOU

Copyright©2014 Huawei Technologies Co., Ltd. All Rights Reserved.

The information in this document may contain predictive statements including, without limitation, statements regarding the future financial and operating results, future product portfolio, new technology, etc. There are a number of factors that could cause actual results and developments to differ materially from those expressed or implied in the predictive statements. Therefore, such information is provided for reference purpose only and constitutes neither an offer nor an acceptance. Huawei may change the information at any time without notice.