AI编译优化--总纲

作者：PAI团队

随着AI模型结构的快速演化，底层计算硬件的层出不穷，用户使用习惯的推陈出新，单纯基于手工优化来解决AI模型的性能和效率问题越来越容易出现瓶颈。为了应对这些问题，AI编译优化技术已经成为一个获得广泛关注的技术方向。这两年来，这个领域也异常地活跃，包括老牌一些的TensorFlow XLA、TVM、Tensor Comprehension、Glow，以及最近呼声很高的MLIR，能够看到不同的公司、社区在这个领域进行着大量的探索和推进。

在过去几年时间里，PAI团队在AI编译优化技术方向投入了比较专注的资源精力，对这个领域也建立起了一定的理解，撰写这个系列文章，旨在达到如下目的：

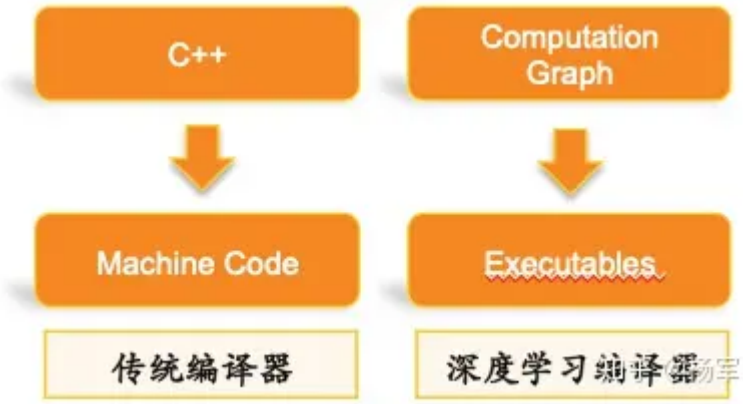
- 对PAI团队在AI编译优化技术方向上的技术投入和取得的结果做一个系统性的总结；
- 阐述我们对AI编译优化整体技术脉络的理解，期望能够对关心AI编译优化技术建设的同行有所启发；
- 对我们关于阿里AI编译优化的技术路径提出一些思考，期望能够抛砖引玉，吸引更多对我们工作感兴趣的同行联系加入我们，一起打造具备阿里特色的AI编译优化技术体系。

本系列文章会由以下内容构成：

本篇会先做一下整体介绍以及行业现状的分析。

为什么需要AI编译器

在上图中将近年的深度学习框架粗略分为三代。几代框架之间有一个趋势，在上层的用户API层面，这些框架在变得越来越灵活，灵活性变强的同时也为底层性能问题提出了更大的挑战。另外一个技术趋势则是系统底层的深度学习的编译器近一段时间也开始活跃起来，这些编译器试图去解决框架的灵活性和性能之间的矛盾。



传统编译器是以高层语言作为输入，避免用户直接去写机器码，而用相对灵活高效的语言来工作，而深度学习编译器的作用相仿，其输入是比较灵活的，具备较高抽象度的计算图，输出包括CPU或者GPU等硬件平台上的底层机器码及执行引擎。

AI编译器的目标是针对AI计算任务，以通用编译器的方式完成性能优化。让用户可以专注于上层模型开发，降低用户手工优化性能的人力开发成本，进一步压榨硬件性能空间。

涉及到性能优化，我们有必要先对一个AI作业执行过程中的性能开销的分布有一个感性的认识，所谓what you can't measure, you can't optimize it. 在这篇paper里，我们针对PAI平台训练workload的性能开销占比有过一个比较细致的分析。考虑到目前我们在AI编译优化里还主要关注单计算设备的计算效率问题，所以我们可以宏观上将单设备上的性能开销拆解为计算密集算子（比如GEMM和Convolution）和访存密集算子（比如Elementwise Add，BN等操作）两部分，关于计算图过于灵活带来的框架开销，我们也统一归类到访存密集算子开销占比中。

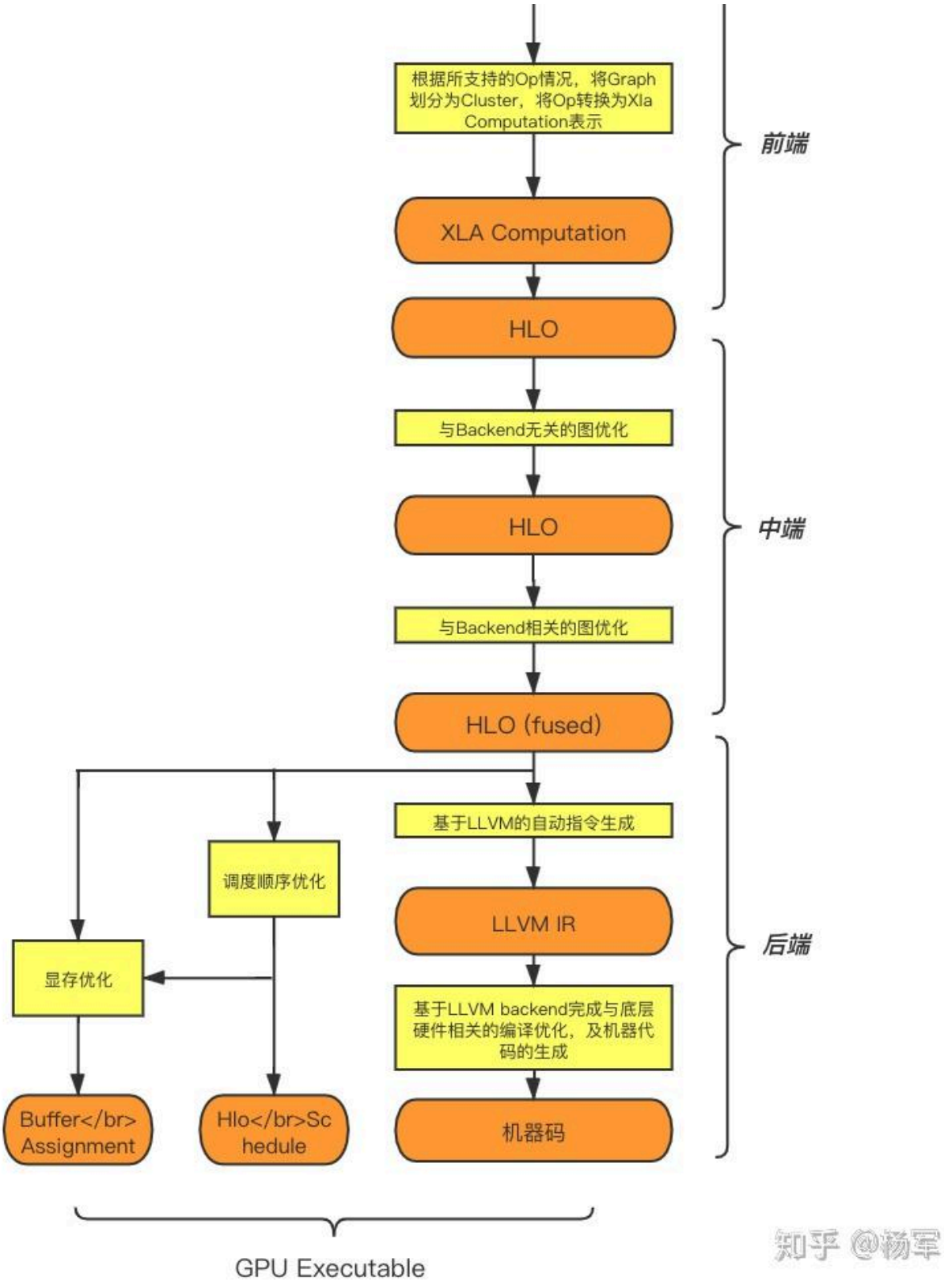
针对不同的性能热点，所需要的优化手段也存在区别。

本章我们首先选取当前两个主流编译框架XLA（针对访存密集算子）和TVM（针对计算密集算子），以及新近非常活跃的MLIR框架为代表（MLIR严格来说不是一个compiler，而是compiler infrastructure，后面会再进行展开介绍），来对业界目前的技术情况做一个概述。

XLA现状简述

原理及主要收益来源

XLA（Accelerated Linear Algebra）是Google于2017年出推出的用于TensorFlow的编译器。XLA使用JIT编译技术来分析用户在运行时创建的 TensorFlow 图，将TensorFlow Op转换成为HLO（High LevelOptimizer）中间表示并在HLO层上完成包括Op Fusion在内的多种图优化，最后基于LLVM完成CPU / GPU等机器代码的自动生成。



上图是XLA的整体架构，其中有两层比较重要的IR（intermediate representation），HLO和底层的LLVM IR。HLO是专门适用于深度学习计算图的IR表示，与TensorFlow Graph的最大区别是其较细的颗粒度，以及有限数量的节点类型；LLVM IR是传统编译器领域的中间层表示，LLVM的中间层架构已经在传统编译器领域得到了充分的验证。注意传统编译器领域一般将LLVM IR之前的部分称作编译器的前端，LLVM IR到底层机器码的这部分称作编译器的后端。而在本文及同系列的后续文章中，会统一将HLO层以及这一层上的图优化工作称作中端，将

TensorFlow Graph向HLO层的转换称做前端，而将HLO层向底层的CodeGen统一称做后端。请读者注意区分。

XLA采取了了一种相对比较朴素的技术路径。对于对自动CodeGen要求较高的计算密集型算子，如MatMul/Convolution等，和TensorFlow一样会直接调用cuBLAS/cuDNN等Vendor Library；而对于除此之外的访存密集型算子，XLA会进行完全自动的Op Fusion和底层代码生成（CodeGen）。除编译本身外，XLA还包含了一套静态的执行引擎。这个静态性体现在静态的Fixed Shape编译（*即，在运行时为每一套输入shape进行一次完整编译并保留编译结果*），静态的算子调度顺序，静态的显存/内存优化等方面，以期望相对于对计算图动态解释执行的TensorFlow，可以获得更好的性能/存储优化结果。

XLA的主要性能收益来源可以概括为如下几个方面：

- 访存密集型算子的Op Fusion收益，这是目前在大多数业务中XLA最主要的收益来源；
- Fixed Shape架构下，TensorFlow计算图中的shape计算相关的子图会在编译时被分析为静态的编译时常量，节省执行时的节点数量。
- HLO层在比TensorFlow Graph的颗粒度上可能存在更大的图优化空间。

此外，XLA的架构也可以方便开发者扩展更多的图优化Pass，包括Layout优化和并发调度优化等等。

访存密集型开销

这里我们举几个实际业务中的timeline片段，希望读者能够对GPU作业上的访存密集型算子的开销占比及其对性能带来的影响有一个直观的了解。

来自某CNN模型训练任务：

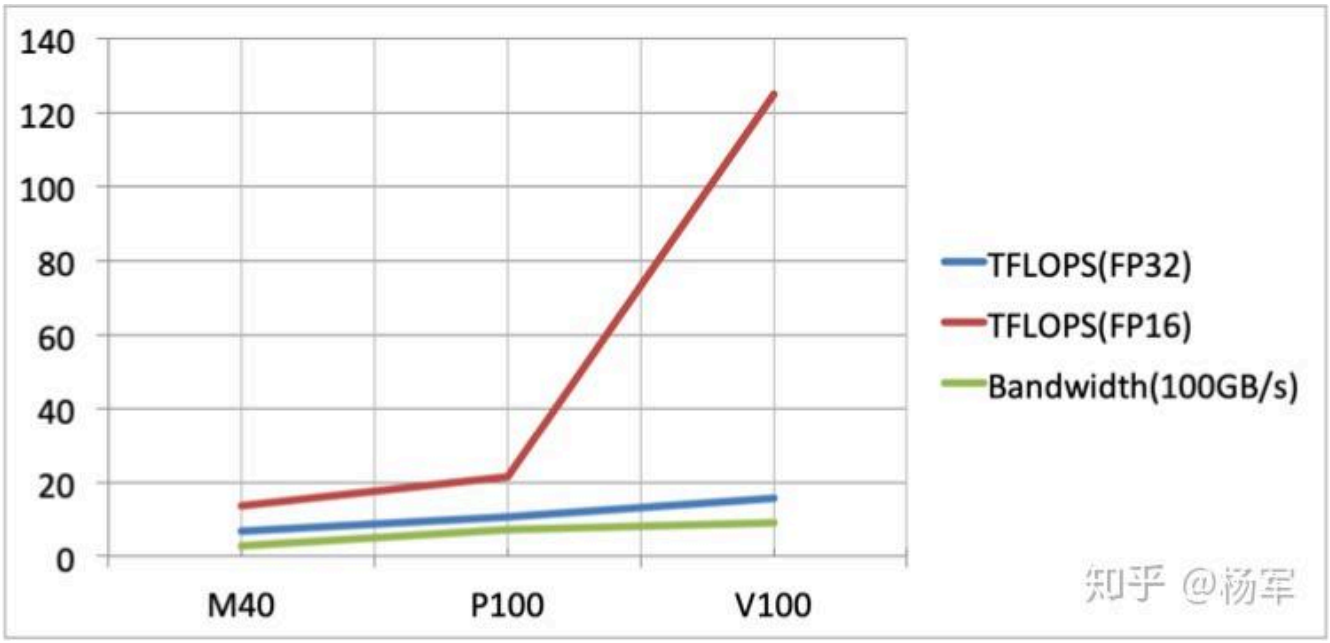
来自某Transformer模型训练任务：

来自一个模型计算图中的LayerNorm计算部分：

来自某RNN训练任务，由于这个业务的计算密度更低，瓶颈转移到host端的Op Launch上，因此kernel之间甚至出现了大量空白：

对于访存密集型算子部分的工作来说，新硬件带来了更大的性能优化空间。下图是NVIDIA近几代硬件的计算throughput和访存throughput的演进趋势，对于计算密集型节点，TensorCore等

专用硬件计算单元的出现大幅提升了硬件的计算能力，灵活多变且不容易在硬件上进行特化优化的细粒度访存密集型算子的开销占比变大，收益空间也相应变大。



用户的使用方式

XLA的目的是帮助用户做通用，透明的一键式性能优化，提升Training / Inference时的Latency / Throughput等，整个过程完全自动。嵌入到宿主TensorFlow里执行的XLA会在原始计算图上自动圈出能够完整支持的子图，不能支持的个别算子可以继续通过TensorFlow的执行引擎来执行，因此对不同的计算图都有比较好的兼容性和可回退特性。从应用场景上，XLA不区分training或者inference，与TensorFlow的良好集成使得它可以实现对用户的完全透明。

优劣势

优势：

- 傻瓜式优化对用户通用透明
- 依托TensorFlow生态，天然支持TensorFlow
- 在GPU后端上，对于访存密集型算子的自动Op Fusion能够在大多数业务中获得明显收益，对于计算密集型算子占主导地位的业务类型，在配合FP16训练/INT8量化Inference之后，也可以在其基础上进一步获得明显性能收益。

劣势：

- 相较TVM/Tensor Comprehension等框架，CodeGen原理比较简单，对计算密集型算子进一步提升性能的发挥空间不大。
- 静态Shape的架构，可以应对单一shape的计算图，或shape变化的枚举数量有限的计算图。但当用户计算图的shape变化范围非常大的时候，应用会存在一定限制（关于这方面的支持，我们已经在积极和社区进行讨论交互，基于最近的演化趋势，propose了一些可能的解决方案，在不远的未来会有进一步的结果进行分享介绍）。

TVM现状简述

TVM 是 MxNet、XGBoost作者陈天奇基于 Halide的思想提出的深度学习自动代码生成工具。它的提出是期望解决计算算子与各种各样的硬件后端之间的 gap，将统一的计算表示通过不同的编译后端来产生对应不同设备的代码。

另外 TVM 还引入了 Learning to Optimize 的 Auto Tuning 机制，在用户给定 schedule 的探索空间之后，Auto TVM 会用实际运行的性能数据作为反馈来指导下一个 schedule 的探索方向，最终得到一个性能非常好的算子实现。

原理及主要收益来源

TVM 基于一种 Tensor Expression Language 的表示方法而设计，其想法最初来源于 Halide，核心思想在于目标代码的计算和调度两个过程分离。

例如给定一个简单的计算表示：

```
C = tvm.compute((n,), lambda i: A[i] + B[i])
```

可以得到如下的 C 语言代码：

```
for (int i = 0; i < n; ++i)
{
    C[i] = A[i] + B[i];
}
```

对其增加额外的调度控制：

```
s = tvm.create_schedule(C.op)
xo, xi = s[C].split(s[C].axis[0], factor=32)
```

则可以得到：

```
for (int xo = 0; xo < ceil(n / 32); ++xo)
{
    for (int xi = 0; xi < 32; ++xi)
    {
        int i = xo * 32 + xi;
        if (i < n)
        {
            C[i] = A[i] + B[i];
        }
    }
}
```

这种对调度控制的分离和抽象，使得相同的计算逻辑在编译过程中可以很方便地针对不同硬件的计算特性进行调整，在不同的后端上生成对应的高效执行代码。经过优化的 TVM 算子能够达到甚至超越通用算子库和专家手动调优代码的性能。

在实际应用场景中，即使是相同类型的硬件也有可能由于参数配置差异对调度控制有不同的需求，如不同型号的 CPU 可能有不同大小的 cache 结构，不同型号的 GPU 在 SM 数量等方面也有所区别等。TVM 的 Auto Tuning 机制提供了自动化的性能探索能力，基于同一套调度模板，Auto TVM 能够通过在不同硬件上进行调优的方式寻找到最适合的模板参数，以达到最佳的性能。

用户的使用方式

TVM 提供了两套不同层级的 API。

Relay 是更高层的图表示 API，包含了很多常见的计算图算子，如卷积、矩阵乘法、各类激活函数等。Relay 中的复杂算子由一系列预定义的 TOPI（TVM Operator Inventory）模板提供，TOPI 中包含了这些算子的计算定义以及调度定义，用户无需关注其中的细节即可很方便地实现自己需要的计算图。

Relay 层 API 还提供了一系列图优化的能力，如 Op Fusion 等。

Relay 计算图在执行时需要首先通过 graph runtime codegen 转换成底层 TVM IR 的表示方式，再进一步编译成对应硬件的可执行代码。

在Relay API之外，用户也可能通过 TVM API 直接在 TVM IR 层进行计算表达，这一步则需要用户同时完成计算表示以及调度控制两部分内容，但这一更贴近硬件的表示层相对 Relay 层来说更为灵活。底层硬件的特殊计算指令可以通过 TVM Intrinsic 调用得以实现。

优劣势

优势：

- Relay IR 与 TVM IR 的分层设计便于不同的优化可以在不同层级实现
- TVM 对计算、调度做了抽象分离，方便多硬件支持
- TVM 非常适合计算密集型算子

劣势：

- 调度控制的撰写仍然需要专家经验，当前未做到真正的自动化(针对这个问题阿里已经与社区在进行合作，在不远的将来对这个问题会进行针对性解决，可以参见我们与社区关于auto-schedule的合作论文以及相关的PR)

MLIR现状简述

MLIR是Google在2019年的CGO上提出的一套compiler infrasructure(注意这个说法，我们在跟进MLIR这条技术线的时候，会发现还是有同学会把compiler infrastructure和compiler这两个不同的概念混淆在一起，这种基础概念的混淆会给一些系统方案的设计以及核心概念的理解带来一定的麻烦。也有人将MLIR称之为Meta Compiler)。从理念和vision上来说，MLIR无疑是宏大的，我们只要看一下其claim的几个核心卖点就可见一斑：

- The ability to represent dataflow graph (such as TensorFlow), including dynamic shapes(_针对XLA的当前局限_), the user-extensible op ecosystem, TensorFlow variables, etc.
- Optimizations and transformations typically done on a such graph (e.g. in Grappler) (_统一优化框架的野心_).
- Representation of kernels for ML operations in a form suitable for optimization(_和下面一条一起，期望将计算密集算子和访存密集算子集中在一套框架里统一打击的野心_).

- Ability to host high-performance-computing-style loop optimizations across kernels (fusion, loop interchange, tiling, etc) and to transform memory layouts of data.
- Code generation “lowering” transformations such as DMA insertion, explicit cache management, memory tiling, and vectorization for 1D and 2D register architectures(_对硬件memory/cache/register的分层描述能力_).
- Ability to represent target-specific operations, e.g. accelerator-specific high-level operations.(灵活的对AI-domain ASIC的描述能力)
- Quantization and other graph transformations done on a Deep-Learning graph(_这一点也很诱人，关于模型层面的优化，如果能够基于MLIR的中间表示层进行统一打击，无疑对于提升行业生产效率是有重要价值的_).

而达到上面的vision及野心的关键点是MLIR的几条核心设计理念：

- A collection of **modular and reusable** software components that enables the **progressive lowering of operations**, to efficiently **target hardware in a common way**

关于**module and reusable**，MLIR整体遵循了open for extension的设计原则，更具体一些，通过创造性地引入了Dialect的设计概念从而使得不同的应用场景可以根据自己的需要对MLIR的表示能力（包括不同抽象层次的算子描述、数据类型以及在相应Dialect层级上施加不同图变换的能力）进行非侵入性的扩充。同时在MLIR的设计体系里，非常强调 round-trippable 的转换语义的保证，再配合上Dialect的扩充能力，就很好地满足了**progressive lowering of operations**的要求。同时MLIR团队表达出很强的野心对于底层算子codegen提供基于Polyhedral技术的优化能力，以达到其**target hardware in a common way**的目标。

之所以强调MLIR作为compiler infrastructre的意义，是因为基于MLIR，可以比较方便灵活地将不同的已经存在优化手段以Dialect的方式进行接入（比如XLA、TVM，甚至TensorRT、nGraph这样的厂商优化工具），与MLIR提供的基础的优化工具进行拼装，形成组合优化的效果。这个价值在我看来，是深远的，因为对于快速迭代的工业界，往往需要的不仅仅是一个好的基础框架，而更需要一个能够将新技术与已存在技术进行无缝集成融合的框架，而MLIR的设计理念就很好地满足了这一点要求。

然而MLIR的设计理念虽然美好，但是其当前仍然处于一个快速演化的阶段，举一个具体的例子。MLIR刚推出时很吸引人的是其claim的high performance codegen的能力， 然而即便到现在为止，MLIR的code base里关于计算密集算子的底层codegen也只是在个别尺寸上取得了还不错的性能，相较于TVM的计算密集算子codegen的支持也还有相当大的距离。 而通过MLIR与不同existing优化手段的桥接，理念上没有问题，并且MLIR社区也已经在推进一些工作（与TensorFlow graph、XLA以及TF Lite的集成），但是大体都还处于experimental的阶段，比如和XLA的接入，乍一看code base里似乎通路是有了，但其实真正想走通MLIR的完整端到端通路目前还仍存在问题，参见这里的一个讨论。以及MLIR对于dynamic shape的支持，目前还仍然处于一个理念完整，细节显著缺失的状态，参见这里和这里的细节讨论，针对dynamic shape，我们也已经向社区propose了一个相对完整的E2E的solution RFC，并在推进实质的工作，预计在不远的将来，我们可以向大家分享更多细节。

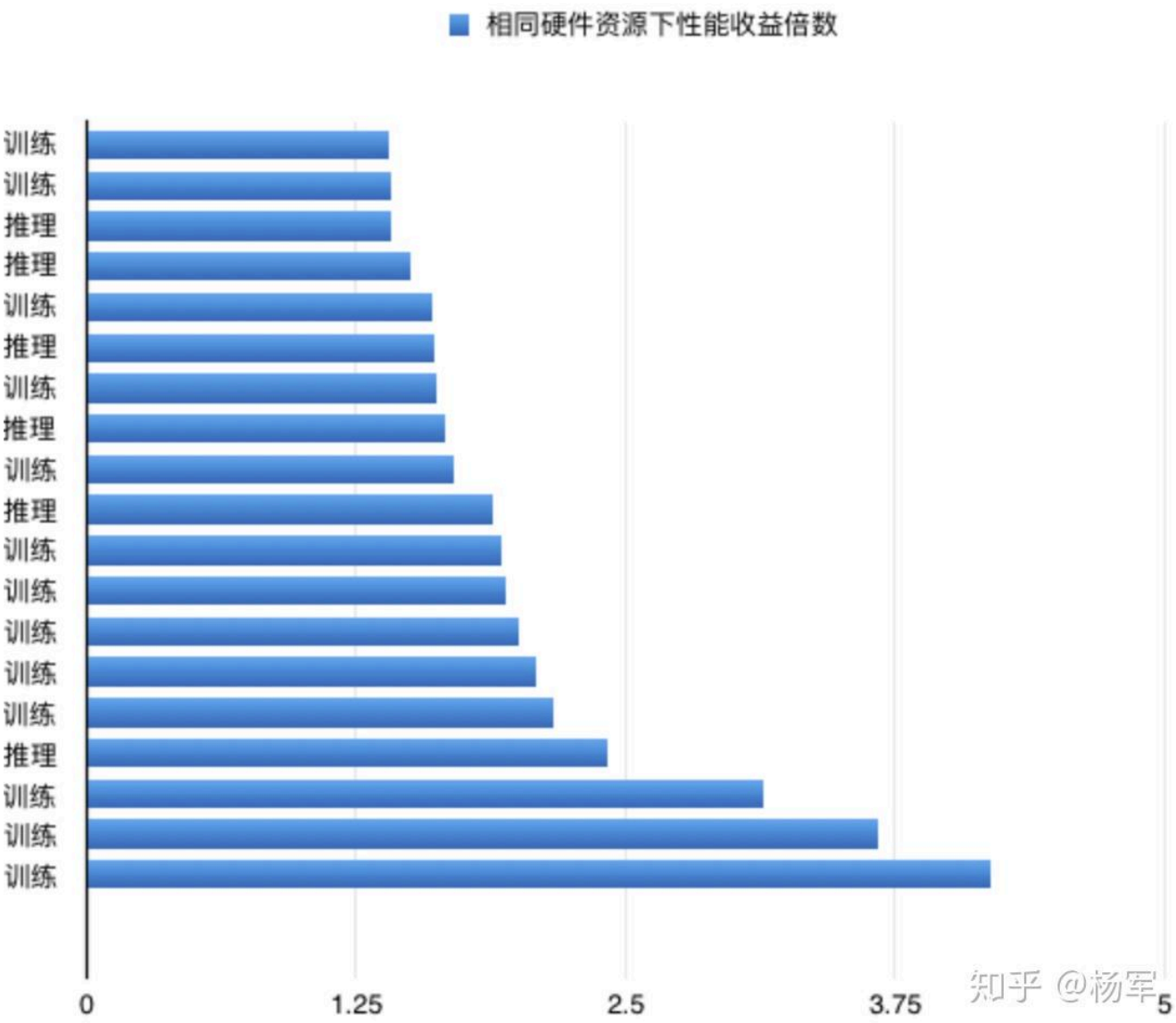
总的来说，对于MLIR，我们的长期态度是乐观的，但具体采纳的节奏是谨慎的。我们也会借助于MLIR的胶水能力来将我们同时在推进的不同方向的优化工作进行整合。MLIR本身模块化的设计思想使得我们在采纳的时候，能够根据自身需要以及对其具体模块稳定性的判断进行灵活的装配，这也是我们对其设计理念比较认可的地方。

PAI的相关工作

PAI团队在过去两年多时间里，围绕AI编译优化主要在XLA、TVM以及MLIR这几条技术线进行了比较深度的投入，也取得了一定的业务结果。XLA上我们提出了具备一定技术原创性的大尺度算子融合技术，并先后历经两版迭代，已经同时应用于训练和推理的生产场景。GPU上针对访存密集算子取得了不错的优化效果。异构硬件上以FPGA定制ASIC逻辑配合编译上做产品化的实操，并积极推动CPU的差异化化探索，全方位使能XLA的编译优化能力赋能框架。

TVM这条技术线我们围绕低精度及新硬件进行了自动codegen的优化探索，部分工作也已经反馈给TVM社区。同时针对我们在实际业务支持过程中发现的问题，我们提出了离线编译系统的想法，基于data-driven的理念构建起了一套离线编译调优的pipeline系统，并与我们的通用推理优化工具PAI-Blade（已经在阿里云进行产品化发布，相关技术细节我们后面也会撰文介绍）已经进行了整合，从而在codegen tuning所需的探索耗时与在线获取编译优化收益之间获得了较好的平衡，用算力来优化算力。持续推动TVM的服务化能力建设，以工具化推动CPU / GPU定制化场景的高性能计算服务能力。同时也在FPGA及异构上面做持续的技术探索，持续化服务弹内异构加速需求。最后，在编译加持的高性能算子库的建设中，TVM也加速了我们cpu / gpu的高性能算子及子图的快速迭代。

下面的图表是基于我们的编译优化工作在GPU上取得的部分训练及推理业务的性能收益，其中X轴的数字为使用相同硬件资源情况下的性能收益：



后面的文章会对我们的工作细节进行更详细的展开，敬请期待。

最后，持续招聘中，欢迎对我们的工作感兴趣的同行、同学、老师^-^邮件联系 muzhuo.yj@alibaba-inc.com，我们虚席以待：）。