AI编译优化--计算密集算子优化

作者：PAI团队

> *进入正题前，还是先打个招聘小广告，欢迎对我们工作感兴趣的同学联系我们，细节参见这里，可以直接邮件muzhuo.yj@alibaba-inc.com。*

本文是AI编译优化系列连载的第三篇，总纲请移步：

https://zhuanlan.zhihu.com/p/163717035

针对计算密集算子，我们的工作包括两大部分：

- 围绕GPU硬件上的低精度算子开展了一系列优化工作，以充分发掘NV新硬件提供的以TensorCore为代表的专用硬件加速单元的计算效率。
- 针对多种硬件设备(GPU/CPU/端侧CPU等)，以更具一般性的方式自动完成计算密集算子的codegen支持，这个工作的细节会在未来的文章中进行着重介绍，在本篇不会涉及，感兴趣的同学可以先通过这篇tech report来了解整体的技术思路。

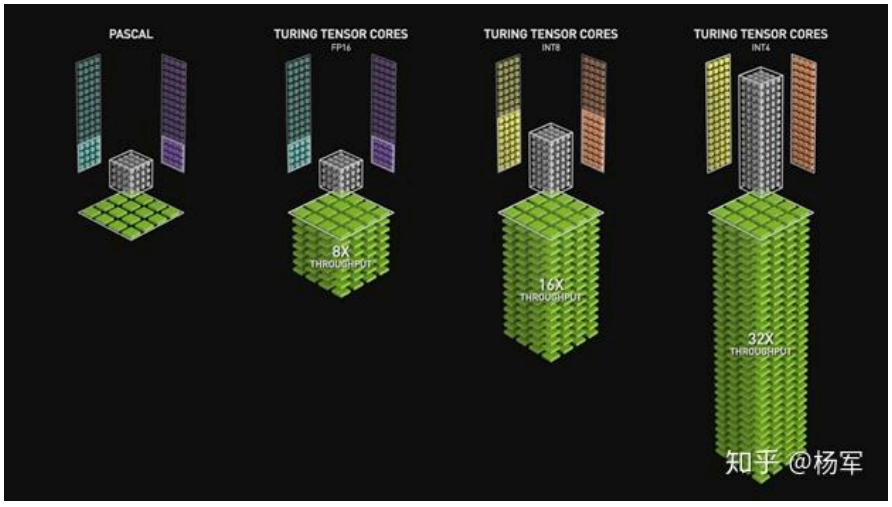
由于最新的AI硬件加速单元都只支持低精度，例如fp16, int8, int4甚至int1，因此想要充分利用这些AI硬件加速单元的前提首先是需要一个低精度的模型，因此在训练侧我们开发了**自动混合精度**，在推理侧我们支持**低精度量化。**

在得到低精度模型之后我们便可以基于硬件厂商提供的高性能计算库（例如针对TensorCore有cuBLAS, cuDNN, TensorRT等）利用AI硬件加速单元的算力大幅提升训练和推理的性能，但是硬件厂商极致手工（汇编）优化需要针对各种情况（例如不同算法、不同尺寸、不同硬件架构）分别进行精细调优，工作量极大，因此很难在所有情况下都确保有高效的实现。所以我们需要更通用更自动化的解决方案来进一步提升AI硬件加速单元的计算效率，我们在TVM框架上实现了**TensorCore AutoCodeGen.** 本文接下来将详细介绍这三部分的工作。

一、自动混合精度

混合精度训练是指在训练过程中在不同的计算中使用不同的数值精度，从而充分挖掘显卡硬件上每一个晶体管的极致性能，目前在PAI上支持的混合精度训练主要是指**FP16和FP32**两个数值精度。在标准TensorFlow训练任务中，variable及tensor的表示精度目前均默认为FP32的，那么为什么要引入FP16的数值精度呢？我们知道对于神经网络训练过程而言，影响速度的两个关键因素是，**计算和访存**：

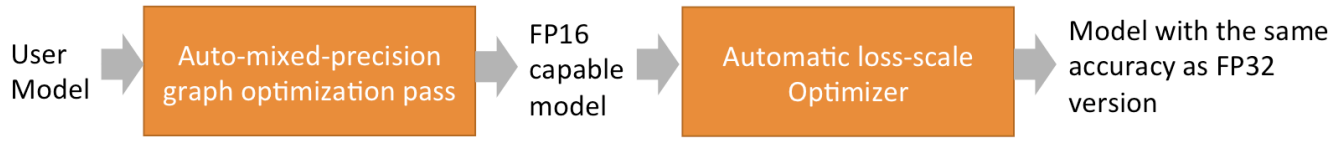
- 从计算而言，NVIDIA在2017年推出了Volta架构，其中重要的模块即为TensorCore（如下图），在TensorCore的加速下，V100在混合精度下的吞吐量比在FP32下有**8X**的加速。TensorCore主要对**matrix-multiply-and-accumulate**类计算进行了加速。基于TensorCore我们可以对 *MatMul*、 *Convolution*等计算原语进行加速，而二者恰好是神经网络中计算量较大的op



- 从访存而言，若tensor的数值精度由FP32变为FP16，那么我们也可以得到理论上**2X**的访存加速。 因此我们引入了混合精度对神经网络的训练进行加速。

自动混合精度训练需要解决的问题分为两个方面：

一是自动将用户定义好的单精度模型转换为高效的混合精度模型（单精度和半精度的混合精度模型），即Auto-mixed-precision graph optimization模块；二是通过模型算法策略解决训练过程中数值精度降低所带来的模型训练精度问题，以帮助用户在不改变训练超参（例如learning rate, optimizer等)的情况下保证模型训练的收敛性，即为automatic loss-scale optimizer模块。



易用性是我们PAI平台在追求高性能、高效率的同时同样重视的另一个关键目标，因为我们的用户，即AI Developer，希望专注于模型和算法本身，不希望被各种复杂的性能优化问题分散精力，这要求我们的性能优化能够作为Turnkey Solution，尽可能对用户透明，避免对用户模型代码的入侵。混合精度训练加速优化同样如此。

因此对于第一个问题，*图转换问题*，我们在计算图优化环节实现了自动的、精细的混合精度转换，对用户完全透明，用户跟正常训练FP32模型一样，无需修改模型代码；同时在计算图优化环节，我们通过相应的性能优化避免混合精度转换过程中伴生的性能不利因素；对于第二个问题，我们实现了比社区对优化更加友好的optimizer封装，辅助用户无缝加入loss scaling策略，使能用户无需关心自动混合精度训练的细节策略，即可基于开关启动混合精度训练，以更好地利用TensorCore的计算加速特性，加速模型训练。

下面我们将依次介绍两个维度的技术解决方案。

自动混合精度图改写

自动混合精度图改写旨在自动帮用户将单精度训练图改写为混合精度训练图，在精度符合模型约束的前提下保证图的高性能。自动图改写可分为以下几个步骤：

White-List:

首先，为了保证模型的高精度以及满足深度学习框架的约束，我们会维护一个white-list，只有white-list中的算子才可能被转换为FP16。

无论对于计算密集算子还是非计算密集算子，我们都期望使得尽可能多的算子转换为FP16或FP16/FP32混合精度进行计算，因为：对于计算密集算子例如Conv2D/MatMul，可以通过混合精度获得TensorCore的硬件加速，而非计算密集算子（即访存密集算子）可以通过FP16获得访存的节省。但在实际图转换中有若干约束使得有些算子不能加入whitel-list。更细节一些，white-list的维护来自两个维度的约束：一方面模型层不适合用FP16进行计算的算法将维护white-list之外，例如reduction类算子（例如batch_norm），exp/log/softmax类算子等；另一方面TensorFlow框架层不支持的FP16的算子也将维护在white-list之外，例如BatchMatMul等；

Cost-Model:

实际上，white-list中的算子并非都应该转换为FP16，计算图中每个节点需不需要转换，除了和这个计算节点本身的算子类型有关，还和这个节点周围其他节点的连接拓扑等因素紧密相关，所以我们需要一个Cost Model来预测每个节点转换为FP16之后对整体性能的影响，从而作出转换与否的判断。Cost-Model可以辅助我们进一步过滤White-List.

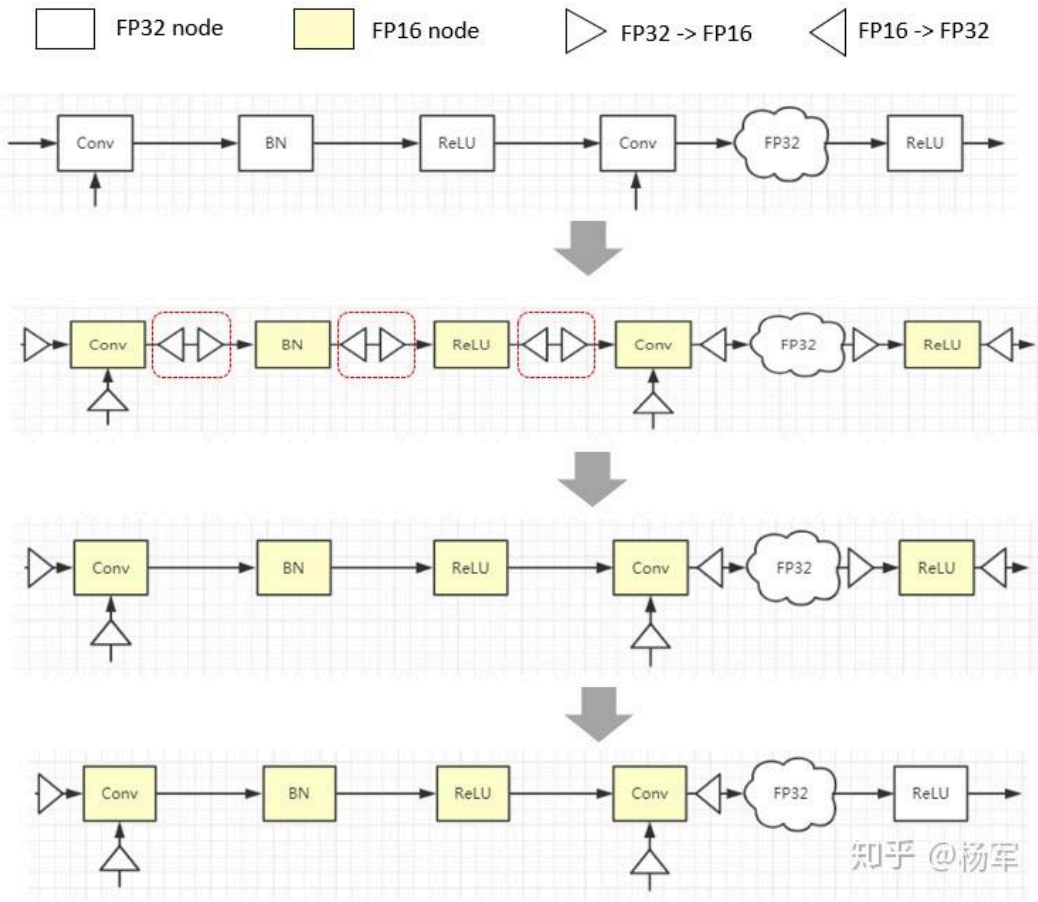
FP16 Propagation:

对于混合精度训练图的改写，我们期望使得图中尽可能多的节点基于低精度进行计算，同时又期望图改写满足模型层的要求，因此我们提出了FP16 Propogation的方案，即以计算密集算子MatMul/Conv2D节点为起点，将FP16算子改写向下游传播，直到white-list之外的计算算子以及需要保留为FP32计算的子图（例如optimizer部分）.

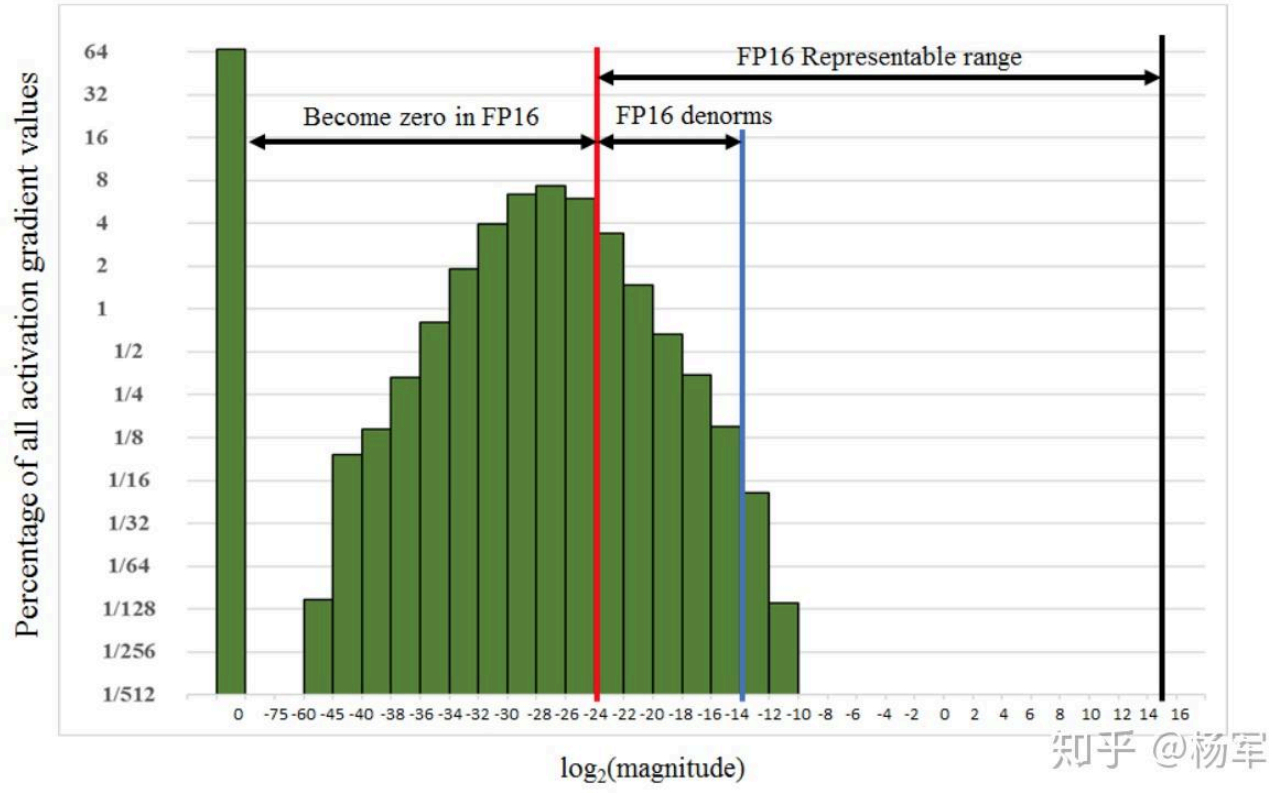
Cast Elimination & Cast Fusion
计算图层面的细粒度改写存在一个不可避免的问题是会产生大量的Cast转换节点，因为将每个节点转换为FP16必然会在输入端插入FP32转FP16的Cast, 输出端插入FP16转FP32的Cast，这些Cast会带来很大的额外开销，因此我们通过Cast Elimination 和 Cast Fusion两个方法来降低所引入的这部分开销，以优化图结构，提供图的运行效率。

Cast Elimination就是将两个连续的相反的Cast节点互相抵消，如下图所示，这种情况在FP16转换后大量存在，如果没有Cast Elimination，这些Cast节点带来的额外开销大部分情况下足以抵消混合精度训练所带来的性能收益。虽然Cast Elimination已经能够消除大部分Cast节点，但并不能完全消除，对于剩余的Cast节点可以进一步通过Cast Fusion使其和前后的节点融合，从而避免带来额外的访存开销和Kernel Launch开销。

计算图自动变换的完整过程的一个示例如下图所示：



自动Loss Scaling实现



SSD训练过程中的gradients分布

由于数值精度的下降，模型的训练过程可能会有一定的精度下降或者不收敛情况。数值精度下降所带来的影响可主要从梯度的数值范围来进行分析。上图所示为SSD模型在单精度训练过程（即当前TensorFlow/PyTorch的默认数值精度）中的gradients分布图，图中红线以左在FP16的表示下均会变为0，即常见的underflow问题，而同时我们观察到FP16有很大一部分表征能力并没有用到，因此一个直观的想法是我们是否可以将gradients放大一定的倍数进行表达。此解决方案可以通过loss scaling策略进行解决，即在loss上乘以一个scaling factor S

，根据链式法则，所有的gradients均会被放大 S 倍，那么我们只需在gradients参与更新计算之前进行unscale（乘以 \frac{1}{S} ）即可。其中对于scaling factor，我们实际上是引入了额外的超参数，若其数值较大，则在后向计算中，gradients可能出现overflow的情况，若其数值较小，则依然不能很好的解决FP16下underflow的问题。对于它的控制，NVIDIA提出了auto scaling策略，其主要想法是，在不溢出的情况下，我们可以使用一个尽量大的scaling factor。

对于Auto Loss Scaling的实现，我们依然遵循的是易用性原则，期望用户以极低的用户成本启动Auto Loss Scaling策略，实现了对算法的框架层封装。在PAI上用户只需要在提交作业的命令行上加一个 -DlossScaling参数即可实现loss scaling.

线上使用方法及加速效果

最终使用自动混合精度训练的方式如下，training_script仍然是用户原先全精度训练的模型代码，无需做任何改动，只需要加上-DautoMixedPrecision参数即自动进行混合精度训练，-DlossScaling是optional的，只有部分模型需要设置loss scaling factor，也可以设置为-DlossScaling='auto', 完全无需关心如何选择一个合适的scaling factor。大体的使用示例如下：

```
PAI -name=$framework_name
    -Dscript=$training_script
    -DautoMixedPrecision=true
    [-DlossScaling=$scale_factor]
```

下表为自动混合精度训练在Public Model中加速效果

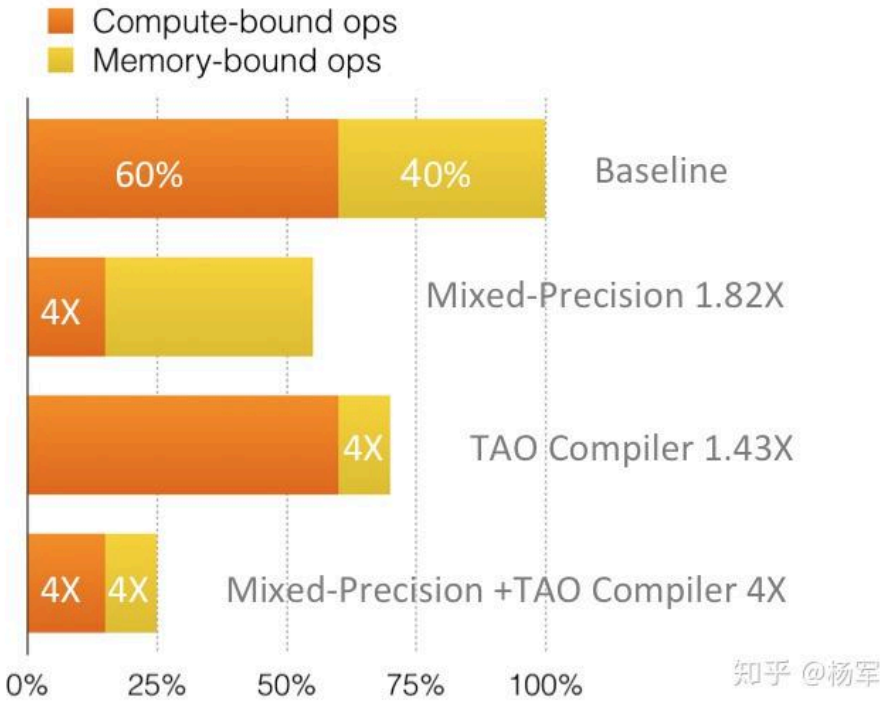
Model type	Description	Speedup
CNN	Resnet50	1.5X
LSTM	Language Model, batch size=512, hidden size=1024	2.3X
GAN	Font Generation	1.4X
Transformer	NMT, batch size=480, hidden size=512	1.6X
Transformer	NMT, batch size=480, hidden size=1024	2.1X
BERT	Finetune, Squad, batch size=10	1.6X

下表为部分弹内用户在真实业务模型开发过程中使用自动混合精度训练获得的加速效果

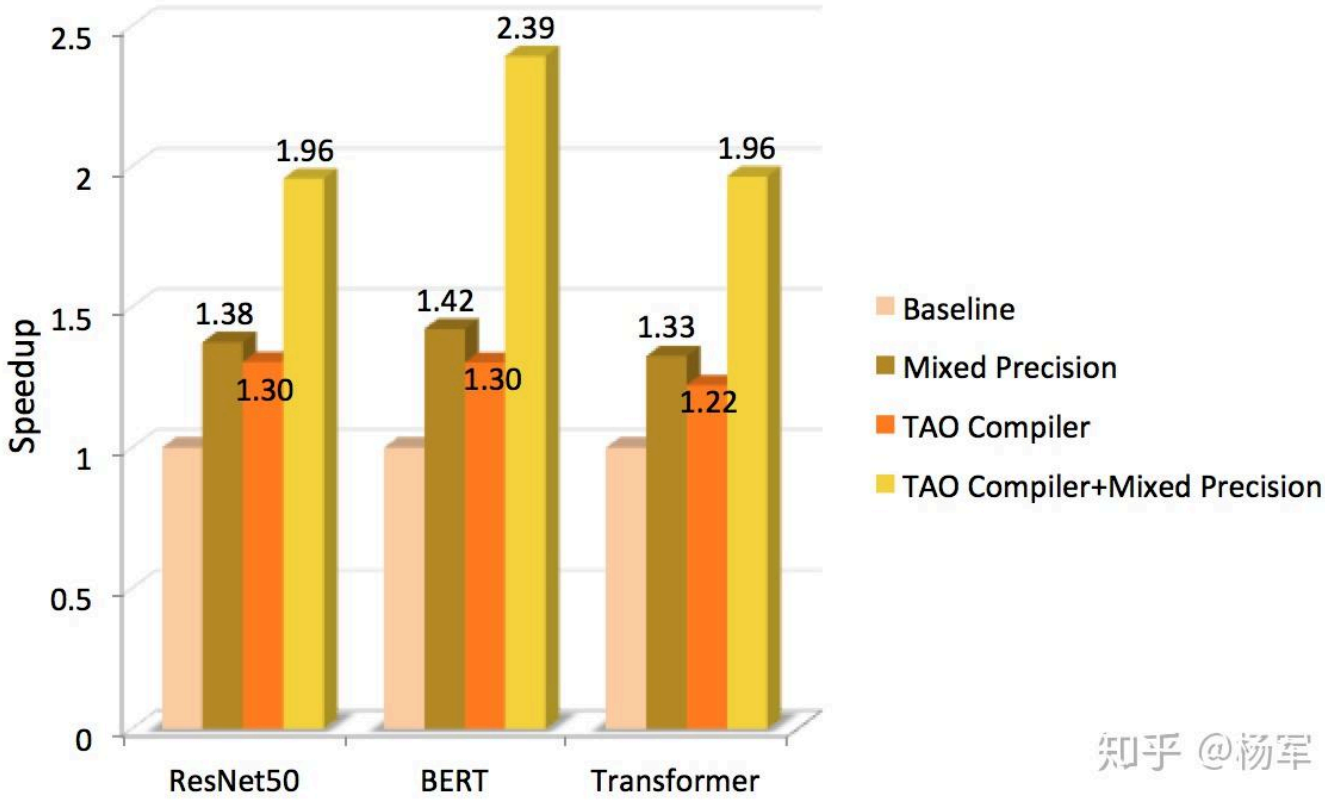
BU	Model	Speedup
[redacted]	DNN	2.4X
[redacted]	DNN	2.9X
[redacted]	CNN	1.5X
[redacted]	CNN	1.4X
[redacted]	CNN	1.5X
[redacted]	Transformer	1.5X
[redacted]	BERT	1.6X
[redacted]	BERT	1.5X
[redacted]	BERT	1.6X

以上case中用户均在未修改训练超参的情况下，完成了与FP32训练一致的训练精度。

混合精度训练与上篇系列文章介绍的访存密集算子优化的结合会有更显著的加速效果。访存密集算子优化的目标为减少访存密集型算子的lauch及访存开销，而混合精度训练的主要目标是减少计算密集型算子的计算开销，因此二者在训练优化加速中是互补的，且二者结合使用具有1+1>2的效果。以下为一个简单的实例，我们不妨假设一个简单的训练任务，其计算密集算子和访存密集算子的开销占比分别为60%和40%，混合精度训练与访存密集算子优化在相应的计算部分均可达到4X加速，则仅使用混合精度训练可达1.82X加速，仅启用访存密集算子优化可达1.43X加速，而二者的结合可达4X加速， $4 > 1.82 * 1.43$ 。如下图所示（图中的TAO Compiler对应于我们内部访存密集算子优化工作的code name，后同）。



下图所示为混合精度训练+访存密集算子优化在三个public models上的加速收益。综合可见，在混合精度训练+TAO Compiler在三个case上均有1+1>2的效果。



由于极致的易用性以及显著的加速效果，自动混合精度在PAI平台上吸引了来自超过X个BU的用户使用，优化的作业数超过XX万个。

二、推理低精度量化

神经网络的训练是一个不断对权重进行细微调整的过程，通常需要浮点精度表示和运算，无法直接用定点数代替（这一两年开始出现了一些将部分网络环节基于定点训练进行加速的研究工作，不过还未进入大规模主流生产阶段，所以在此先不展开）。但在模型预测阶段，由于深度神经网络模型本身具有强鲁棒性，能够很好地应对一定强度的输入噪声，排除数据中无关信息的干扰，因此如果将低精度运算视作一种噪声来源，神经网络模型应能够给出相对准确的结果。我们通常认为大多模型在预测阶段使用8-bit定点数进行存储和运算足以达到精度要求。

INT8量化（Quantization），又被称为定点化，即对已训练好的FP32模型直接进行INT8量化处理，用8-bit数据进行神经网络的存储和计算，不仅能够明显减小模型文件尺寸（大约可以压缩到原P32模型大小的25%），而且能够有效提高在线预测时的访存和计算效率。

量化策略

将已训练好的浮点精度模型直接进行定点化处理，转换成相应的INT8模型。过程中，需对模型参数进行定点化处理，并将计算图中的FP32计算节点替换成相应的INT8计算节点。下面将详细介绍具体的量化策略。

（1）对称/非对称量化

对称均匀量化的策略，即SCALE ONLY的均匀量化方式，借助一个scale值将浮点数对称量化到表示范围 [-127,127]，使得浮点数 0.0 映射到定点数0的位置，量化后的定点数为有符号数。即：

INT8_value = round(FP32_value * scale)

具体地，在对模型参数进行定点化处理时，将首先统计浮点数据的待量化范围 [-m, m]

，其中 m=max(abs(FP32_value)) 然后将浮点数m映射到定点数127的位置，即 scale=127/m 。

非对称量化的策略，即有bias的量化方式：

INT8_value=round(FP32_value*scale+bias)

一般情况下，待处理的浮点数据并非对称分布，此时非对称的量化方式能够更好地利用定点数的表示能力，而对称的量化方式则会浪费一定INT8表示。但是，在进行等价的定点化运算时，非对称的量化方式相比于对称量化，其计算次数会明显增加。

经实验验证，在大多数常见的卷积神经网络模型中，是否采用非对称量化对最终量化效果的影响较小。因此，为获得更理想的加速效果，会倾向于选择对称量化的策略。

（2）在线/离线量化

由于是对一个已经训练好的FP32模型直接进行定点化处理，因此在实际量化的过程中，模型参数是固定的，可以直接通过寻找量化边界m进行对称量化。但在INT8模型实际预测过程中，每次前向计算的中间结果（即activation）与其输入数据密切相关，无法像模型参数一般直接进行一次性量化，这里介绍两种activation量化模式：在线量化模式、离线量化模式。

在线量化策略，即在预测过程中根据实际的activation数据范围实时计算scale，进行在线的量化，本文中记为在线量化模式（Online）。

离线量化策略，即需要用户预先给定一个校正数据集，基于该数据集统计预测时activation的数据分布，预先计算出合适的scale值并存储在INT8模型中，本文中记为离线量化模式（Offline）。

比较两种量化模式，在线量化模式的量化效果相对稳定，但在线预测时会存在计算scale的开销；离线量化模式在预测时无需实时计算scale，却需要预先提供一个具有代表性的校正数据集，该数据集选取的合适与否会直接影响到最终的量化效果。

（3）量化精度调整

大部分已训练好的分类模型在进行INT8量化后精度损失不明显，如VGG16、ResNet50、InceptionV3等网络，但部分精简网络（如MobileNet等）或其它任务模型（如Bert等）在进行INT8量化后存在一定的精度损失。在该类情况下，可以通过对网络预先进行权重调整（Weight Adjustment）等方式，以减小模型量化后的精度损失。

加速效果

下表为1080Ti上量化前后的加速情况（batchsize=16），表中模型量化前后精度损失均小于1%。

	Baseline	Online INT8	Offline INT8
VGG16	40.15ms	19.58 (2.05X)	12.85 (3.12X)
ResNet50	28.86ms	10.88ms (2.65X)	5.36ms (5.38X)

下表为T4上量化前后的加速情况（batchsize=16），表中模型量化前后精度损失均小于1%。

	Baseline	Offline INT8
VGG16	72.58ms	10.22 (7.10X)
ResNet50	51.04ms	7.08 (7.21X)
InceptionV3	64.92ms	17.85 (3.64X)

如前述，部分模型量化后存在一定的精度损失，如下表所示。下表中数字为精度结果，括号中数字为耗时或加速比，其中“INT8”表示直接进行INT8量化的结果，“INT8+WA”表示加入权重调整的INT8量化结果，由表中数字可见，精度损失明显减小。

	Baseline	INT8	INT8+WA
Bert	85.47% (561.02ms)	85.17% (1.71X)	76.37% (1.76X)
MobilenetV1	71.01% (31.15ms)	70.51% (2.77X)	69.43% (2.77X)
MobilenetV1_050	59.20% (9.71ms)	56.48% (1.84X)	51.84% (1.84X)

三、TVM TensorCore AutoCodeGen

背景介绍

前文提到了TensorCore 是 NVIDIA 从 Volta 架构的 GPU 开始引入的技术，是一种 GPU 的片上专用硬件加速器。GPU 利用 TensorCore 可以快速完成对规定尺寸矩阵的乘累加操作，极大地提升了乘加运算的吞吐率。

TensorCore是一种可编程的硬件加速器，每个TensorCore完成4x4x4(在NV最新一代的A100架构里这个规格升级为8x4x8)的小矩阵乘，在 CUDA 中 TensorCore 通过 WMMA api 来调用，这是一种特殊的 warp-level 编程模型，由同一个 warp 中的不同线程联合完成给定矩阵计算。因此在生成的 CUDA 代码中，同一个 warp 内的所有线程都需要用相同的参数去完成 WMMA API的调用，这与常规的 CUDA 编程模型有较大的的差别。

WMMA API形式上虽然没有太多特殊的地方，但是有些TensorCore相关的特殊输入给编程以及自动代码生成造成了一定的难度，比如fragment的定义需要指定矩阵是matrix_a, matrix_b 或者accumulator，row major还是col major， 以及warp tile size的m, n, k；load/store访问memory buffer的index也比较特殊，因为这是一个以warp为单位的操作，一个warp内的thread对应的index需要保持相同。

TVM社区在去年加入了TensorCore CodeGen的支持，为WMMA API添加了相应的Intrinsic，如*tvm_load_matrix_sync*, *tvm_mma_sync*等等。Fragment是一种特殊的register buffer，且区分matrix_a, matrix_b, accumulator, 因此新增了对应的Memory Scope，构造出warp level的schedule就可以通过tensorization把对应的操作替换成TensorCore Intrinsic，从而实现TensorCore CodeGen.

但是写warplevel schedule并应用tensorization其实是有一定难度和时间开销的，用好Intrinsic需要对WMMA API有一定的了解；从长远来看，在schedule层面的工作越少越好，尤其是随着auto schedule的发展，未来将不需要额外写schedule；短期来看我们也希望有一个统一的GPU Matmul Schedule，增加可维护性，共同的优化可以只做一遍，统一的Schedule也方便在整个探索空间寻优，既包括TensorCore也包括非TensorCore优化项 ，因为也有可能会出现非TensorCore kernel比TensorCore kernel快的情况。

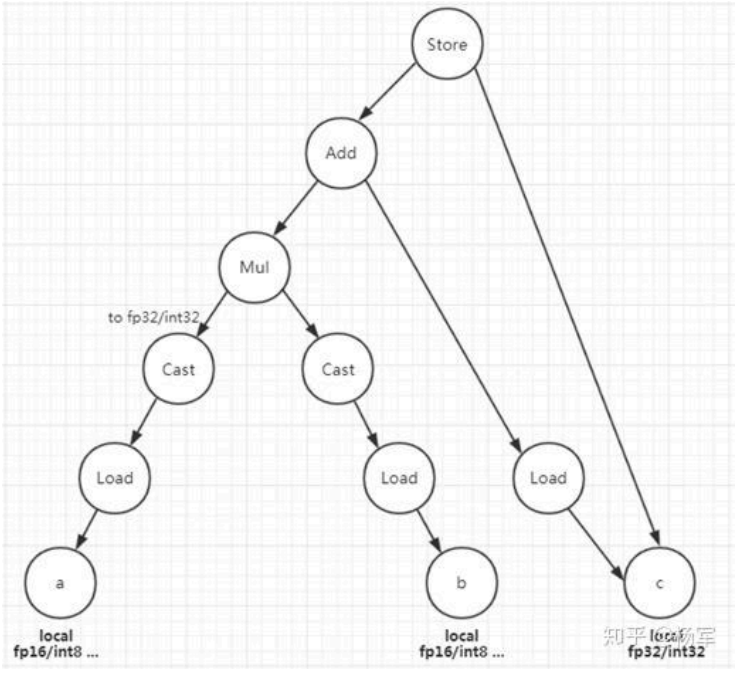
实现方案

我们的解决方案就是直接从普通的thread-level schedule生成TensorCore Code，普通的schedule是指用于生成CUDA code的schedule，不是所有的普通schedule都能符合TensorCore的要求，需要warp tile满足一定的要求。

我们的解决方案在某种程度上可以看作是一种Auto Tensorization，我们的做法是通过IR Pass将语法树中的一些子树转换成对应的TensorCoreIntrinsic，如下图所示

Pattern Matching

在这个方案里Pattern Matching是基础，需要转换的子树就是通过Pattern Matching找出来的，最主要的是如下图所示的这个对应mma_sync的子树，根节点是Store，叶子节点是a, b, c 三个local buffer，其中a, b是低精度，c是全精度，a, b load之后cast成全精度后相乘再累加到c buffer

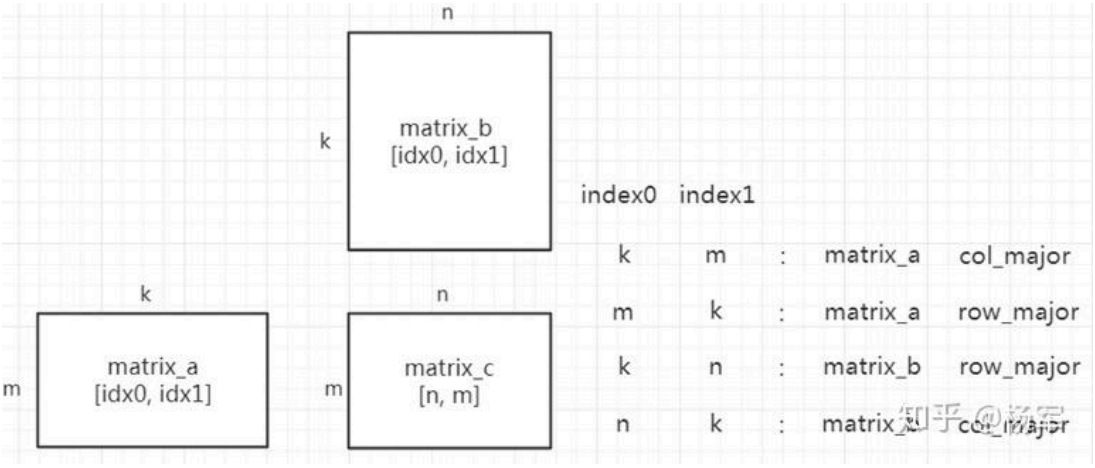


mma_sync子树匹配成功后，a, b, c即对应fragment，再匹配load和store子树就很容易，分别如下二图所示

Pattern Matching是第一步，如果匹配不成功，说明这个语法树不是矩阵乘计算，因此直接跳过TensorCore，生成普通的CUDA代码。

Matrix Identification

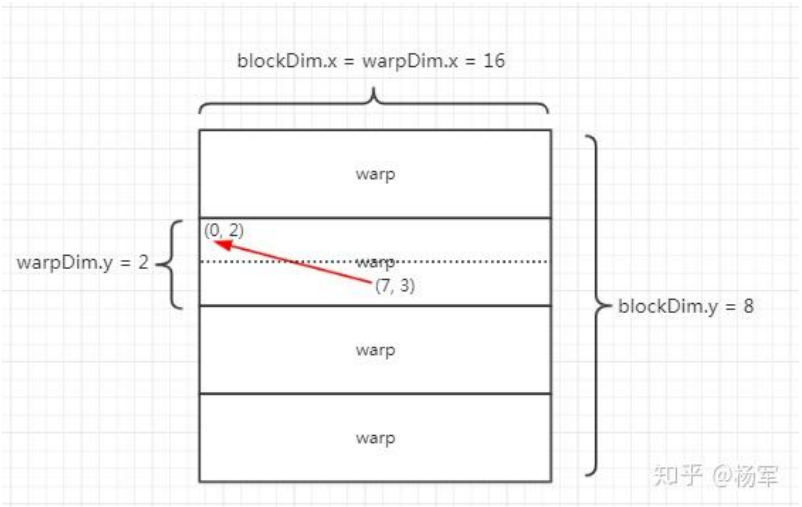
前面在背景介绍里提到了wmma fragment的定义需要指定matrix_a, matrix_b, 以及是row_major还是col_major，这些high level的矩阵信息在语法树层面是不具备的，而这些信息即使让用户（schedule开发者）输入，用户也未必分得清，因为cuBLAS的matrix_a/matrix_b以及row_major/col_major与常规的表示相反。我们的方案通过对比输入矩阵的访存indices和输入矩阵的axis之间的关系自动判断每个输入矩阵是matrix_a/matrix_b以及row_major/col_major. 如下图所示，输出矩阵C的axis分别是(n, m)，则matrix_a必然有一个access index和axis m相对应，matrix_b必然有一个access index和axis n相对应，由此可以判断出某个输入矩阵是matrix_a还是matrix_b. 然后进一步根据和axis m和axis n对应的access index的dimension即可判断出输入矩阵是row_major还是col_major. 例如：如果某个输入矩阵dimension 1的access index是axis m，那么这个输入矩阵是matrix_a + col_major, 还可以进一步检查确认这个输入矩阵dimension 0的access index必须是reduce axis k.



Thread Index Unification

TensorCore是warp level的编程模型，因此访存时一个warp内的各个thread需要访问相同的地址，这要求对应的thread index保持相同，而CUDA编程模型里每个thread天然具备unique的thread index. 针对这个问题我们实现了ThreadIndex Unification功能，把同一个warp内部的所有thread的index都变成相同的，也就是起始thread的index.

如下图所示，一个warp 32个thread，2x16，可以看出来起始thread的threadIdx.x一定是0，所以只要把所有thread的threadIdx.x全置为0就可以。起始thread的threadIdx.y是2的整数倍，这里称之为warpDim.y, 也就是说所有的threadIdx.y向下取整到warpDim.y的整数倍。也就是先除以warpDim.y再乘以warpDim.y。但是warpDim.y并不存在，这里利用到了warpDim.x = blockDim.x来计算，因为blockDim.x是可以直接获取的，而warpDim.x*warpDim.y = 32，所以warpDim.y = 32/warpDim.x = 32/blockDim.x. ThreadIndex Unification需要确保只在load store fragment的子树上修改，不能在整个语法树上，因为其他部分，比如load store shared memory依然是正常的thread level编程模型。



Loop Scaling

前面Pattern Matching匹配出来的乘累加子树虽然对应mma_sync, 但实际上这两者不是等价的，计算量相差非常大，*mma_sync*是一个warp做了16x16x16次乘加计算，而后者是每个thread做了一次乘加计算，因此一个warp做了32次乘加，因此，mma_sync对应于后者的16x16x16/32, 也就是128倍，即

$$\text{“wmma::mma\_sync(c, a, b, c)”} = \text{“c = float(a)*float(b) + c”} \times (16x16x16/32)$$

因此，为了等价替换，需要将语法树中对应的loop scale 128倍。可是应该选择哪些loop去scale呢？我们这里的做法是根据fragment register的访存index所对应的的IterVar，下面是两段等价的normal CUDA code和TensorCore Code的对比，注意其中TensorCore code把两个循环次数分别是8和16的两个loop scale成了1.

```
//Normal CUDA code
 for (int k_inner_inner = 0; k_inner_inner < 16; ++k_inner_inner) {
      for (int j_c = 0; j_c < 8; ++j_c) {
          compute_local[j_c] = (compute_local[j_c] + ((float)(A_shared_local[k_inner_inner] * B_shared_local[((k_inner_inner * 8) + j_c)])));
      }
 }

//TensorCore code
for (int k_inner_inner = 0; k_inner_inner < 1; ++k_inner_inner) {
    for (int j_c = 0; j_c < 1; ++j_c)
        wmma::mma_sync(compute_local[0], B_shared_local[0], A_shared_local[0], compute_local[0]);
    }
}
```

实现TensorCore AutoCodeGen还有很多其他挑战，例如怎么从语法树获取warp tile size等等，这里不再一一赘述，有兴趣的同学可以参考TVM codebase里的src/pass/http://tencore_core.cc(注：TVM社区大约在去年年末开始推进了一次大的基础IR升级动作，升级之后，Auto TensorCore codegen pass因为IR兼容性的原因暂时被disable，后续我们会考虑在最近推进的TVM auto-schedule的工作里支持TensorCore的codegen寻优，所以暂时没有推进这个pass的升级动作)

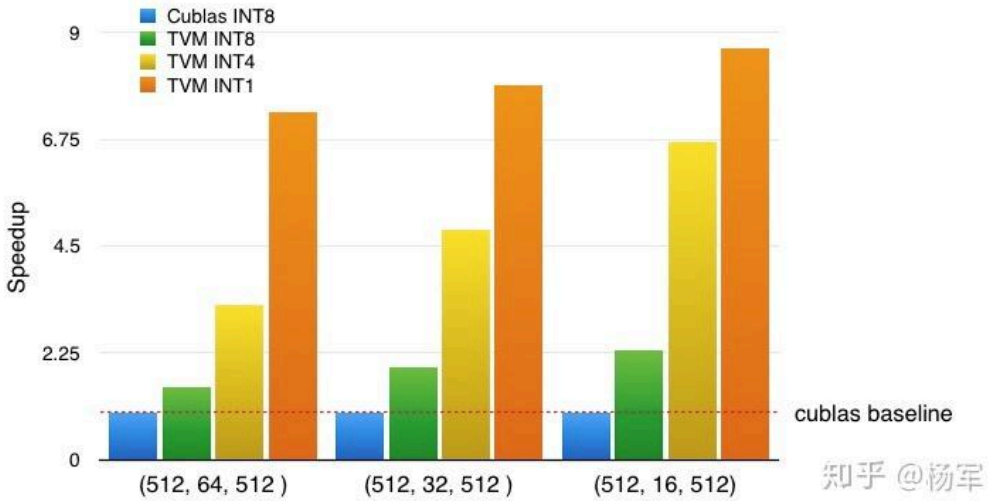
由于是直接从Normal MatMul schedule生成TensorCore code，因此原先MatMul schedule里的优化都同样适用，例如：Auto tune tiling sizes，Vectorized load/store for higher bandwidth utilization，Double buffer to hide memory load latency，Storage align to reduce bank conflicts of shared memory等等，不需要针对TensorCore引入额外的优化。

性能数据

下表是在V100上的FP16性能，在这些尺寸（来自于实际业务模型）上相比于NVIDIA cublas的TensorCore kernel有20%~50%的性能提升

M, N, K	cuBLAS TensorCore	TVM TensorCore	speedup
512, 16, 512	7.7470us	5.2570us	1.47X
512, 32, 512	8.0140us	6.0220us	1.33X
512, 64, 512	8.7530us	6.2390us	1.40X
512, 128, 512	9.0290us	7.1610us	1.26X
256, 256, 256	6.9380us	4.5930us	1.51X
1024, 32, 512	8.3320us	6.3770us	1.30X
2048, 32, 512	9.0640us	7.5070us	1.21X

下图是在T4上的int8/int4/int1性能对比, 在这几个尺寸上Int8 kernel相比于cuBLAS有50% ~ 130%的加速，cuBLAS当时没有提供int4和int1的kernel，但是通过TVM CodeGen很容易添加相应的支持，相比于int8 kernel，性能有很显著的提升，可以使得模型量化的实验不被block.



四、总结

本文介绍了PAI团队针对计算密集算子的三个方面的工作：

- 自动混合精度训练 -- PAI用户无需改动一行代码即可在原有全精度模型上实现fp16混合精度训练，从而利用TensorCore的强大算力。
- 推理低精度量化 -- 通过对网络预先进行权重调整（Weight Adjustment）等方式，显著降低了模型量化后的精度损失。
- TVM TensorCore AutoCodeGen -- 使用普通的TVM MatMul Schedule就能够自动生成TensorCore kernel，相比于cuBLAS TensorCore kernel有显著的加速（fp16 kernel加速20%~50%, int8 kernel加速50%~130%，此外支持生成int4/int1 kernel。

这里介绍的内容涵盖了PAI团队在计算密集算子编译优化相关的一部分工作，后面我们还会对最新的工作进展进行单独撰文介绍。当前我们还正在推进一些更有意思也更具挑战性的工作，涉及到模型和系统编译的联合优化，也欢迎感兴趣的同学联系我们，一起来进行AI编译技术的建设打造。