# Vitis AI 用户指南

UG1414 (v3.5) 2023 年 9 月 28 日

本文档为英语文档的翻译版本，若译文与英语原文存在歧义、差异、不一致或冲突，概以英语文档为准。译文可能并未反映最新英语版本的内容，故仅供参考，请参阅最新版本的英语文档获取最新信息。

AMD 自适应计算矢志不渝地为员工、客户与合作伙伴打造有归属感的包容性环境。为此，我们正从产品和相关宣传资料中删除非包容性语言。我们已发起内部倡议，以删除任何排斥性语言或者可能固化历史偏见的语言，包括我们的软件和 IP 中嵌入的术语。虽然在此期间，您仍可能在我们的旧产品中发现非包容性语言，但请确信，我们正致力于践行革新使命以期与不断演变的行业标准保持一致。如需了解更多信息，请参阅此链接。

# 目录


第 1 章：Vitis AI 概述.....4
- 按设计进程浏览内容.....4
- 功能特性.....6
- Vitis AI 工具概述.....6
- Vitis AI 容器.....11
- 最低系统要求.....12

第 2 章：模型最优化.....13
- 概述与安装.....13
- 剪枝.....14
- TensorFlow (1.15) 版本 - vai_p_tensorflow.....21
- TensorFlow (2.x) 版本 - vai_p_tensorflow2.....24
- PyTorch 版本 - vai_p_pytorch.....27

第 3 章：量化模型.....38
- 概述.....38
- Vitis AI 量化器流程.....39
- TensorFlow 1.x 版本 (vai_q_tensorflow).....40
- TensorFlow 2.x 版本 (vai_q_tensorflow2).....56
- PyTorch 版本 (vai_q_pytorch).....77
- ONNX Runtime 版本 (vai_q_onnx).....101

第 4 章：编译模型.....108
- Vitis AI 编译器.....108
- 使用基于 XIR 的工具链执行编译.....109
- VAI_C 用法.....135

第 5 章：部署和运行模型.....136
- 使用 VART 进行编程.....136
- 使用 VART 进行 DPU 调试.....137
- 自定义运算符工作流程.....141
- 使用 VOE 进行编程.....163
- 多 FPGA 编程.....167
- 使用 WeGO.....168

第 6 章：模型剖析.....186
- Vitis AI Profiler.....186

附录 A：错误代码....194

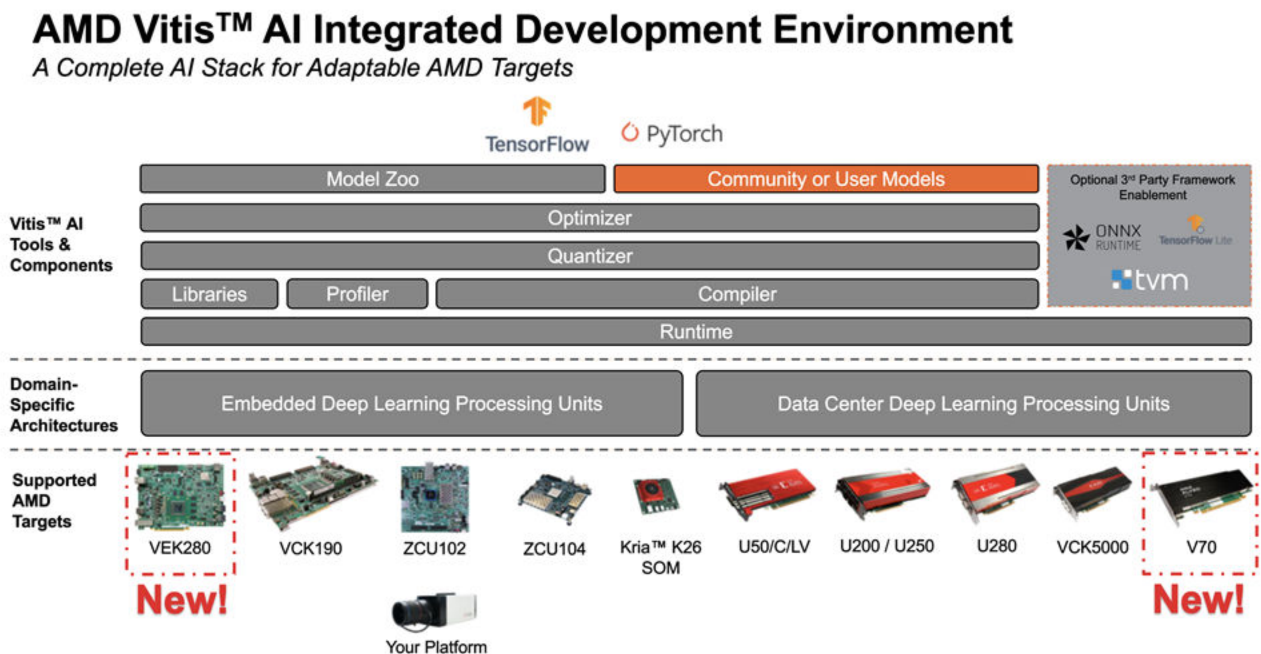
附录 B：附加资源与法律声明....212
- 查找其他文档....212
- 支持资源....212
- 参考资料....213
- 修订历史....213
- 请阅读：重要法律声明....215

第 1 章

# Vitis AI 概述

AMD Vitis™ AI 开发环境可在 AMD 硬件平台上加速 AI 推断，包括边缘器件和 AMD Versal™ 加速器卡。这一综合性框架包括最优化 IP 核、多用途的工具、功能强大的库、多元化的模型和直观的设计示例。Vitis AI 以高效和易用性为核心，使其得以在 AMD SoC 和自适应 SoC 上成功解锁 AI 加速的全部潜能。Vitis AI 开发环境将底层可编程逻辑的繁复细节加以抽象化，从而帮助不具备广泛 FPGA 知识的用户轻松开发深度学习推断应用。

图 1：**Vitis AI 集成开发环境**




**注释：**从 v2.5 起不再提供 Caffe 支持。如需了解 Caffe 支持相关信息，请参阅《Vitis AI 2.0 用户指南》。

# 按设计进程浏览内容

AMD 自适应计算文档按一组标准设计进程进行组织，以便帮助您查找当前开发任务相关的内容。您可以在设计中心页面上访问 AMD Versal™ 自适应 SoC 设计流程。您还可以使用设计流程助手来更深入了解设计流程，并找到特定于预期设计需求的内容。

- 机器学习和数据科学：将机器学习模型从 PyTorch、TensorFlow 或其他热门框架导入 AMD Vitis™ AI，然后对其有效性进行最优化和评估。本文档中适用于此设计进程的主题包括：
  - 第 2 章：模型最优化
  - 第 3 章：量化模型

- 第 4 章：编译模型
- 第 5 章：部署和运行模型
- 第 6 章：模型剖析

# 功能特性

Vitis AI 具有如下功能特性：

- 支持主流框架和最新模型，能够完成多样化的深度学习任务。
- 提供一整套经过预优化的模型，这些模型可直接部署在 AMD 器件上。
- 提供强大的量化器，支持模型量化、校准和微调。对于高级用户，AMD 提供了可选 AI 优化器，它能以可接受的精度损失对模型执行高达 90% 的剪枝。
- 提供逐层分析，以帮助识别瓶颈。
- 提供统一的高层次 C++ 和 Python API，从而最大程度提升从边缘到数据中心的可移植性。
- 自定义高效且可缩放的 IP 核，满足诸多应用的需求，如吞吐量、时延和功耗等方面的需求。
- 自定义高效且可扩展的 IP 核，以满足您不同的应用需求，并最优化吞吐量、时延和功耗等关键指标的性能。

# Vitis AI 工具概述

## Vitis AI Model Zoo

Vitis AI Model Zoo 是经过微调的精选深度学习模型集合，专为加速在 AMD 平台上部署 AI 推断而设计。它包括多种多样的应用，如 ADAS/AD、视频监控、机器人技术和数据中心。它为开发人员配备了强大的工具和经最优化的模型，以充分解锁深度学习加速的优势。

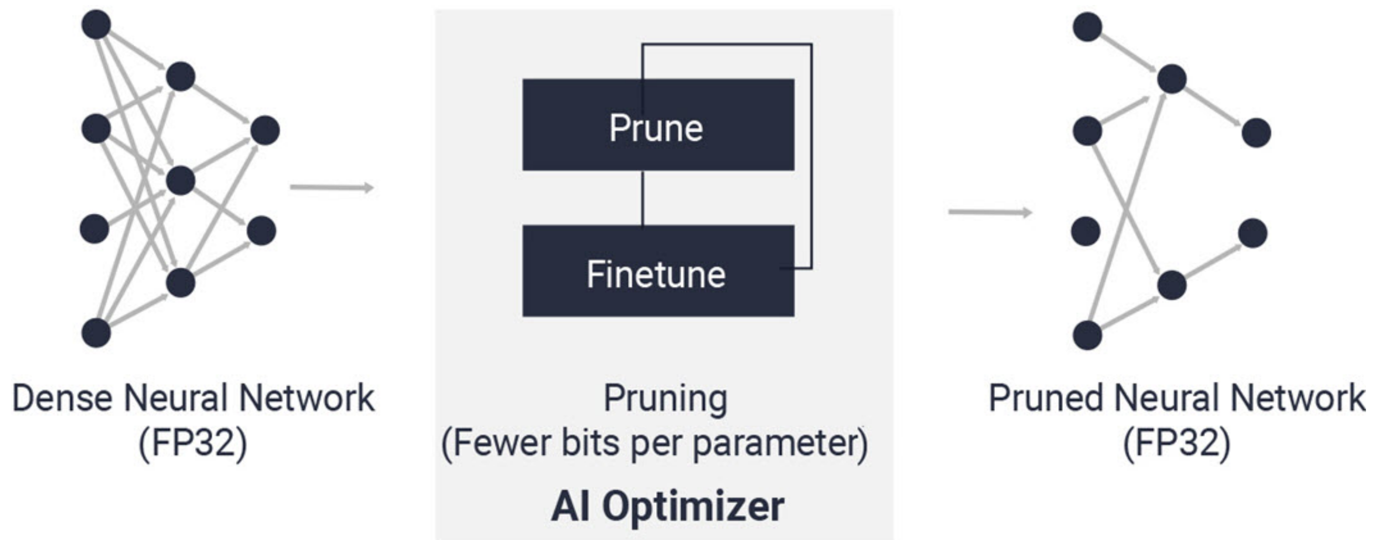
如需了解更多信息，请参阅 GitHub 上的 Vitis AI Model Zoo。

图 2：**Vitis AI Model Zoo**

## Vitis AI 优化器

凭借世界领先的模型压缩技术，可令模型复杂性降低 5 倍到 50 倍，同时确保最低限度的精度降级，如此惊人成就于您而言唾手可得。

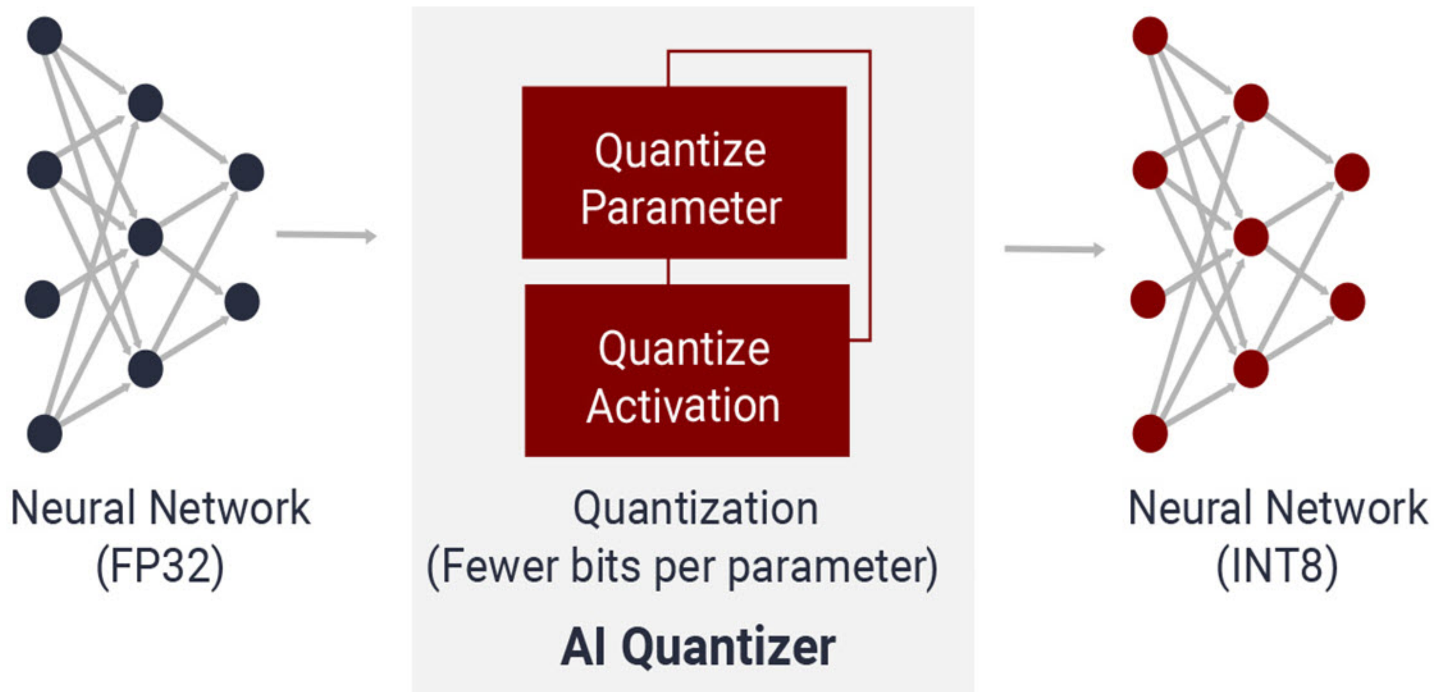
图 3：**Vitis AI 优化器**




## Vitis AI 量化器

Vitis AI 量化器会将 32 位浮点权重和激活转换定点格式（如，INT8），这样即可显著降低计算复杂性，同时保留预测精度。与浮点模型相比，这种变换所产生的定点网络模型对存储器带宽的要求更低，从而提高了处理速度和能效。

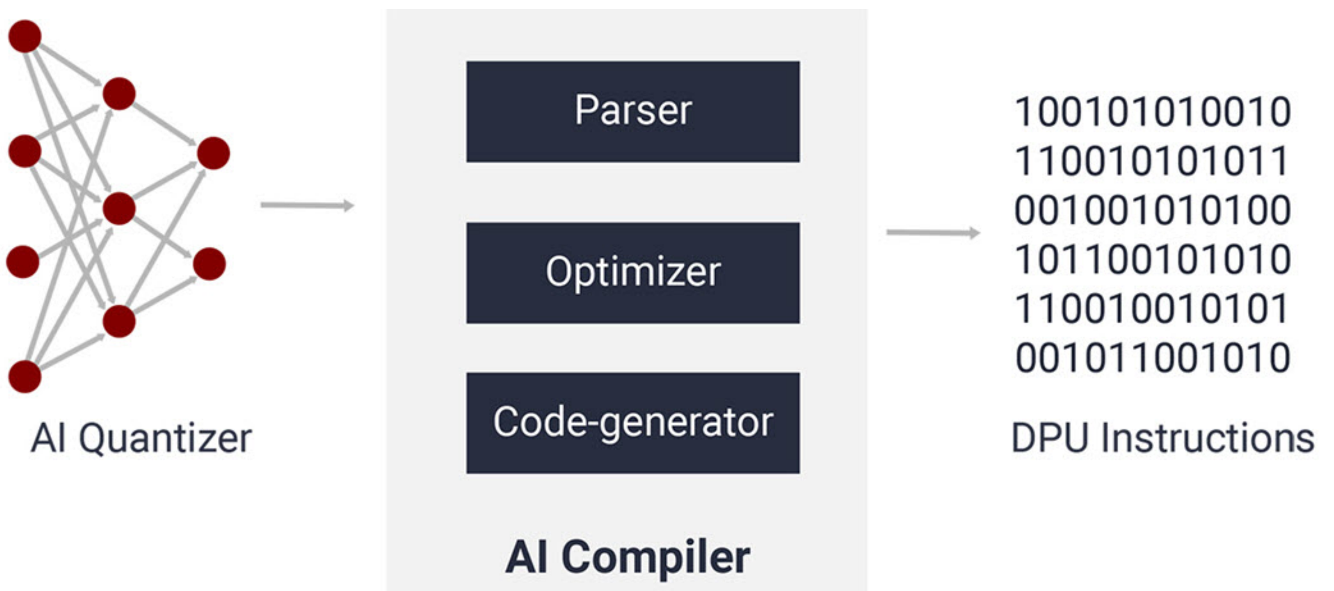
图 4：**Vitis AI 量化器**




## Vitis AI 编译器

Vitis AI 编译器可将 AI 模型映射到高效的指令集和数据流模型。它还可尽可能执行复杂的最优化操作，例如，层融合、指令调度和复用片上存储器。

图 5：**Vitis AI 编译器**

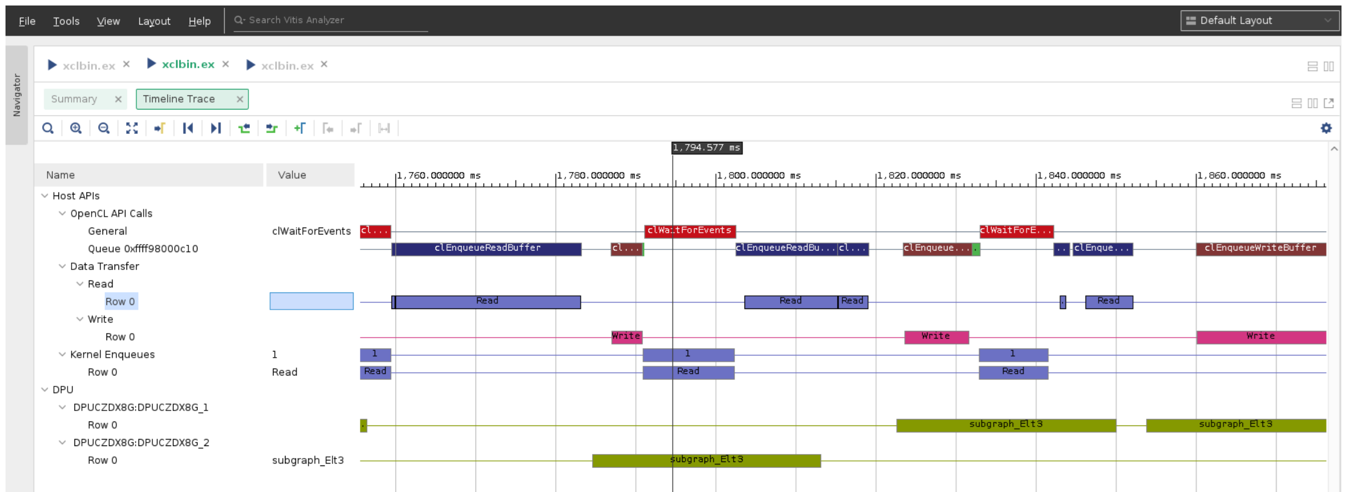


## Vitis AI Profiler

Vitis AI Profiler 剖析器可用于对 AI 应用进行性能分析和可视化，以在不同器件之间查找瓶颈并分配计算资源。它使用方便且无需更改任何代码。它可追踪函数调用和运行时，也可收集硬件信息，包括 CPU、DPU 和存储器使用率。

Vitis AI Profiler 剖析器是一款强大的工具，可用于对 AI 应用进行细致的分析和可视化，识别不同器件之间的瓶颈并以最优方式分配计算资源。其用户友好的界面简单易用，无需修改代码。该剖析器能够追踪函数调用和运行时间，并收集包括 CPU、DPU 和存储器使用率在内的关键硬件信息，从而为开发者提供全面的见解，帮助他们进行高效的性能调优。

图 6：**Vitis AI Profiler**

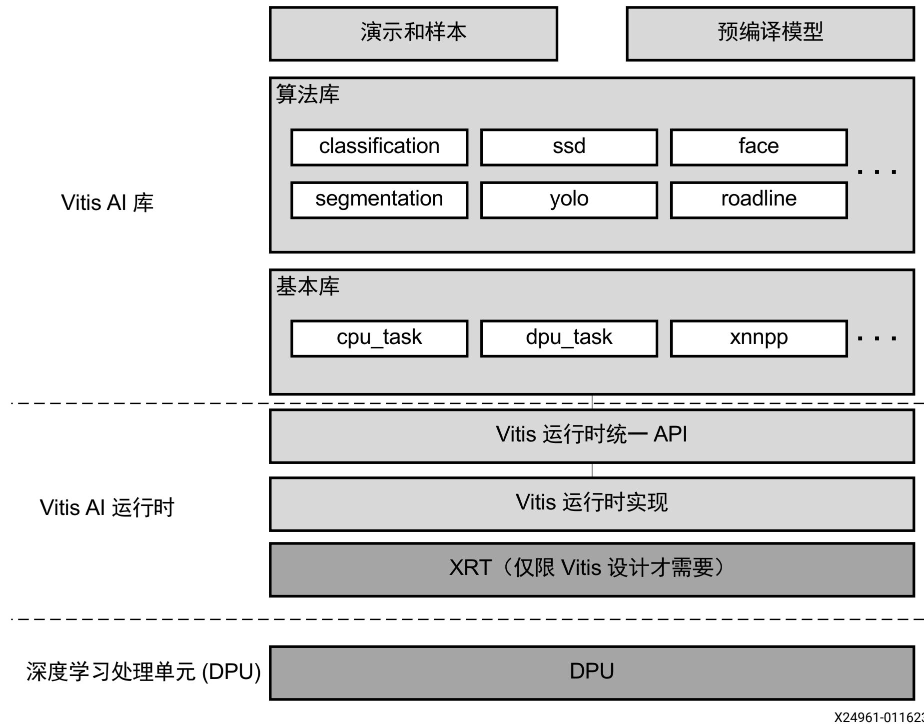



## Vitis AI Library

Vitis AI Library 是一组高层次库和 API，专为利用 DPU 高效执行 AI 推断而设计。这个功能强大的框架与 Xilinx Runtime (XRT) 无缝集成，基于 Vitis AI 运行时构建，包含 Vitis 运行时统一 API，可确保流畅、统一的体验。

Vitis AI Library 通过封装诸多高效且高质量的神经网络，提供易用且统一的接口。它可简化深度学习神经网络的部署，对于不具备深度学习或 FPGA 经验的用户也是如此。Vitis AI Library 使您能够专注于开发自己的应用，而不是底层硬件。

图 7：**Vitis AI Library**




如需了解有关 Vitis AI Library 的信息，请参阅《Vitis AI Library 用户指南》(UG1354)。

## Vitis AI 运行时

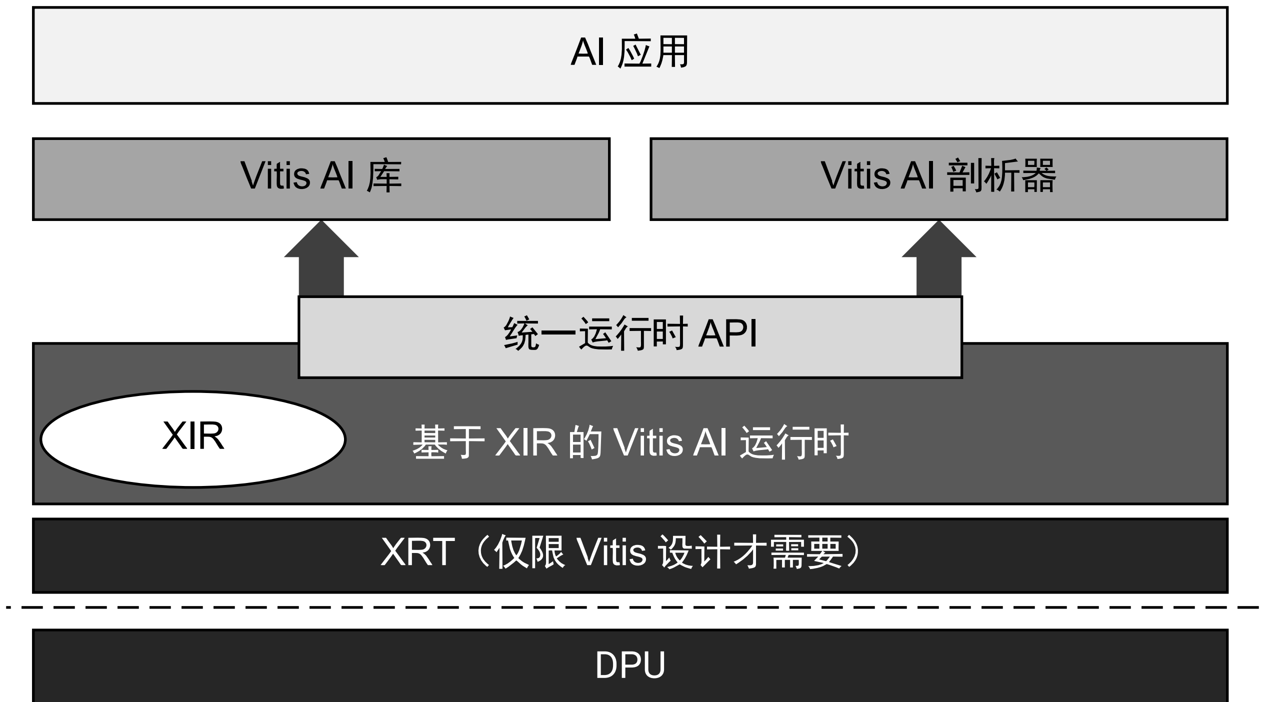
各种应用可借助 Vitis AI 运行时来访问统一的高层次运行时 API，从而实现无缝、高效的数据中心到边缘部署。这种连贯的接口可简化应用部署进程，确保数据中心与边缘环境之间的平稳过渡。

AI 运行时 API 的功能如下所述：

- 向加速器异步提交作业
- 从加速器异步收集作业
- C++ 和 Python 实现
- 支持多线程和多进程执行

下图显示了此 VART 框架。XIR 是 AMD 中间表示形式，是 Vitis AI 中用于表示神经网络运算符的内部 IR。

图 8：**VART 栈**




# Vitis AI 容器

Vitis AI 3.5 发行版使用容器来分发 AI 软件。该版本包含以下组件：

- 工具容器
- 在公开 GitHub 上提供了示例
- Vitis AI Model Zoo

**工具容器**

工具容器包含：

- 通过 Docker Hub 分发的容器
- 统一的编译器流程包括：
    - 面向 DPUCZDX8G（边缘）的编译器流程
    - 面向 DPUCVDX8G（边缘）的编译器流程
    - 面向 DPUCV2DX8G（边缘和数据中心）的编译器流程
- 预构建的 conda 环境用于运行框架：
    - 基于 TensorFlow 的流程：`conda activate vitis-ai-tensorflow`

- 基于 TensorFlow2 的流程：`conda activate vitis-ai-tensorflow2`
- 基于 PyTorch 的流程：`conda activate vitis-ai-pytorch`

  **注释：**对于 PyTorch 中的 WeGO 工作流程，请激活以下 Conda 环境：

  ```
  conda activate vitis-ai-wego-torch
  ```

- Versal 运行时工具

# 最低系统要求

以下 URL 包含利用 Vitis AI 集成设计环境所需的系统要求：

https://xilinx.github.io/Vitis-AI/3.5/html/docs/reference/system_requirements.html

第 2 章

# 模型最优化

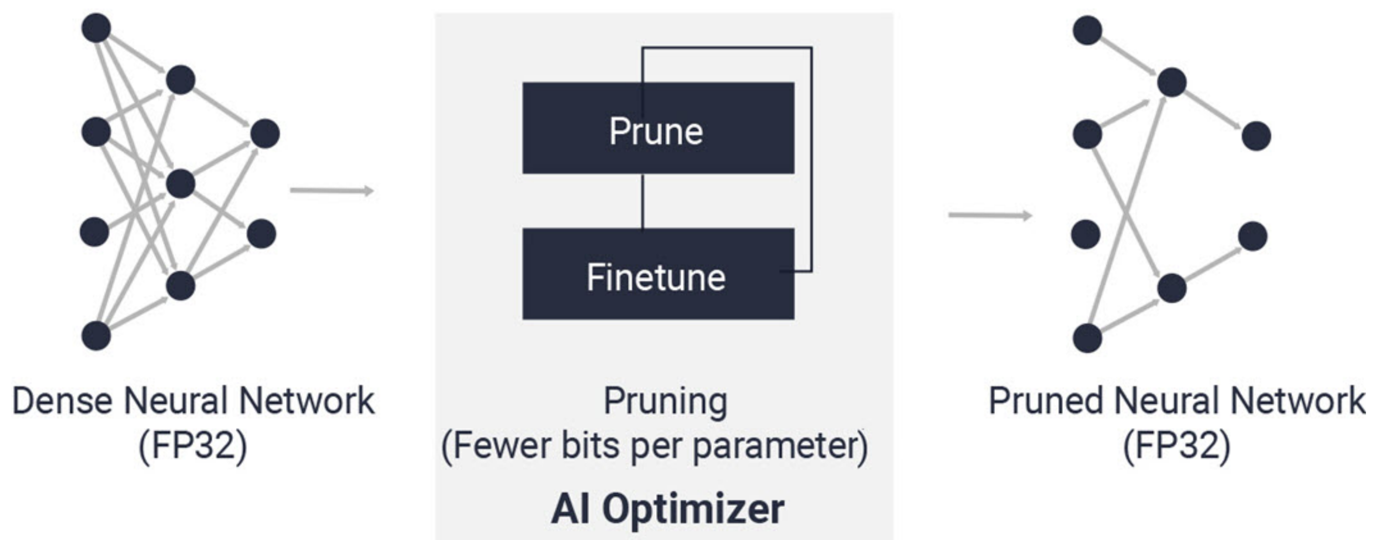
## 概述与安装

### Vitis AI 优化器概述

AMD Vitis™ AI 是 AMD 开发套件，用于在 AMD 硬件平台上进行 AI 推断。机器学习推断是计算密集型流程，需要大量存储器带宽以满足低时延和高吞吐量要求。

Vitis AI Optimizer（优化器）支持对神经网络模型进行最优化。当前，Vitis AI 优化器仅包含一项工具，称为“pruner”（剪枝器）。Vitis AI 优化器用于移除神经网络中的冗余内核，减少推断的总体计算成本。使用 Vitis AI 优化器剪枝的模型可量化并部署在 AMD FPGA、SoC 或自适应 SoC 器件上。

图 9：**VAI 优化器**




Vitis AI 优化器支持 TensorFlow 和 PyTorch。下表中列出了这些对应的工具名称（_p_ 表示剪枝）。

表 1：**Vitis AI 优化器框架和工具名称**

| 框架 | 工具名称 |
|---|---|
| TensorFlow | vai_p_tensorflow (TF1.15) 和 vai_p_tensorflow2 (TF2.x) |
| PyTorch | vai_p_pytorch |

## 受支持的框架和功能特性

下表着重展示了 Vitis AI 优化器支持的功能特性和框架：

表 2：**Vitis AI 优化器支持的框架和功能特性**

| 框架 | 版本 | 功能特性 | | |
|---|---|---|---|---|
| | | 迭代 | 单步 | OFA |
| PyTorch | 支持 1.4 到 1.13 和 2.0 | 支持 | 支持 | 支持 |
| TensorFlow 1.15 | 支持 1.15 | 支持 | 不支持 | 不支持 |
| TensorFlow 2.x | 支持 2.4 - 2.12 | 支持 | 不支持 | 不支持 |

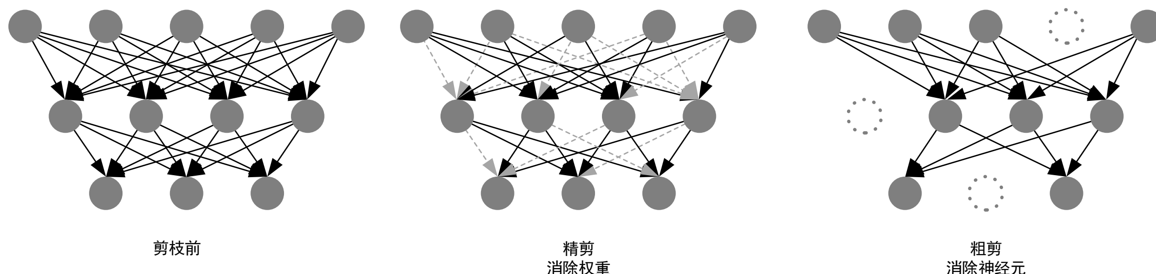
**注释：**Vitis AI 2.5 和更高版本不支持 Caffe 和 Darknet。要使用这些框架，请使用更低版本。

# 剪枝

## 概述

神经网络通常过度参数化，具有大量冗余。剪枝是消除冗余权重同时尽可能使精度损失保持处于低位的进程。

图 10：**低精度剪枝和高精度剪枝**




行业研究带来了多项有助于降低神经网络推断成本的技术。这些技术包括：

- 高精度剪枝
- 低精度剪枝
- 神经架构搜索 (NAS)

## DPU 支持的剪枝方法

Vitis AI 支持以下剪枝方法。这些方法适用于不同的架构，如下表所示：

表 3：不同 **DPU** 架构支持的剪枝方法

| DPU IP | 高精度 | 低精度 | NAS |
|---|---|---|---|
| DPUCZ | 不支持 | 支持 | 支持 |
| DPUCV | 不支持 | 支持 | 支持 |
| DPUCV2 | 不支持 | 支持 | 支持 |
| DPUCA | 不支持 | 支持 | 支持 |

# 高精度剪枝

凭借高精度剪枝，可将对输出影响最小的权重设为 0，以便从推断计算图中跳过或移除对应的计算。这会得到稀疏矩阵（即，具有大量 0 元素的矩阵）。高精度剪枝能够以适度的精度缩减来达成高压缩率。但能够实现高精度稀疏性的硬件加速器必须采用完全自定义的流水打拍实现，或者采用通用的“处理引擎矩阵”类型的加速器并添加专用硬件搭配权重跳过和压缩技巧。

Vitis AI 稀疏性剪枝器能在 M 值的每个连续块内为多种 N:M 稀疏性模式实现高精度稀疏剪枝算法。剪枝算法会沿着输入通道维度对权重值进行剪枝。以 M 个权重为一组，剪枝器会设置 N 个权重（最小值为 0）。M 的典型值是 4、8 或 16，N 等于 M 值的一半，这样即可达成 50% 的高精度稀疏性。

Vitis AI 稀疏性剪枝器支持为卷积层和完全连接的层设置权重和激活稀疏度。激活的稀疏度可为 0 或 0.5。当激活的稀疏度为 0 时，权重的稀疏度可设为 0、0.5 或 0.75。当激活的稀疏度为 0.5 时，权重的稀疏度只能设为 0.75。

稀疏性剪枝步骤如下所示：

1. 生成稀疏模型
2. 微调稀疏模型
3. 导出稀疏模型

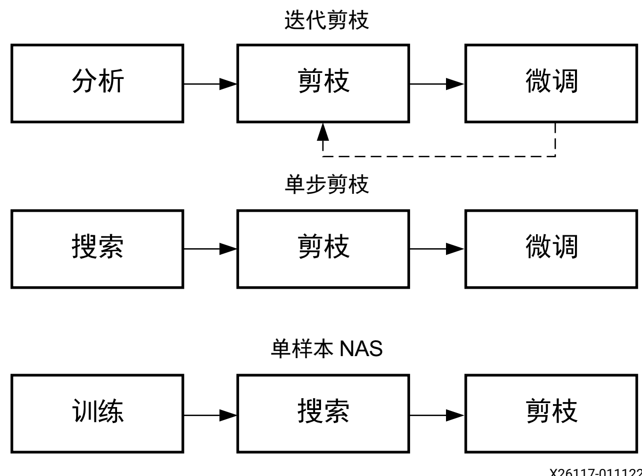
# 低精度剪枝

低精度剪枝也称为通道剪枝，其目的是对通道进行剪枝而不是对个别权重进行剪枝。其结果是一个计算图，其中，剪枝算法会对给定层的一个或多个卷积内核进行剪枝。例如，剪枝前具有 128 条通道的卷积层在剪枝后可能只需计算 57 条通道。

通道剪枝十分适合硬件加速，可通过任何推断架构来实现。然而，由于必须始终对整个内核进行剪枝，因此可实现的总体剪枝率低于高精度实现。

低精度剪枝始终会降低原始模型的精度。重新训练（微调）会对剩余权重进行调整以恢复精度。此技巧对于利用传统卷积的大型模型（例如，ResNet 和 VGGNet）很有效。但对于 MobileNet-v2 之类的逐通道卷积模型，剪枝后的模型精度会显著下降，即使剪枝率很低也是如此。

图 11：三种低精度剪枝方法的工作流程




## 迭代剪枝与单步剪枝的对比

对神经网络进行剪枝的两种主要方法是迭代剪枝和单步跳入剪枝，每种方法都提供了独特的策略，在保持准确性的同时实现模型的稀疏度。迭代剪枝法在保持精度的同时逐步修剪模型参数，通过多次迭代达到所需的稀疏度。相比之下，单步跳入剪枝能快速识别并微调最有潜力的子网络，因此，该方法不仅是实现模型稀疏度的有效选择并且潜在精度很高。

下表中显示了这两种方法的对比。

表 4：**迭代剪枝对比单步剪枝**

| 条件 | 迭代剪枝 | 单步剪枝 |
|---|---|---|
| 要求 | - | 网络中采用 BatchNormalization |
| 耗时 | 超过单步剪枝 | 少于迭代剪枝 |
| 是否需要重新训练 | 必需 | 必需 |
| 代码组织 | 评估函数 | 评估函数<br>校准函数 |

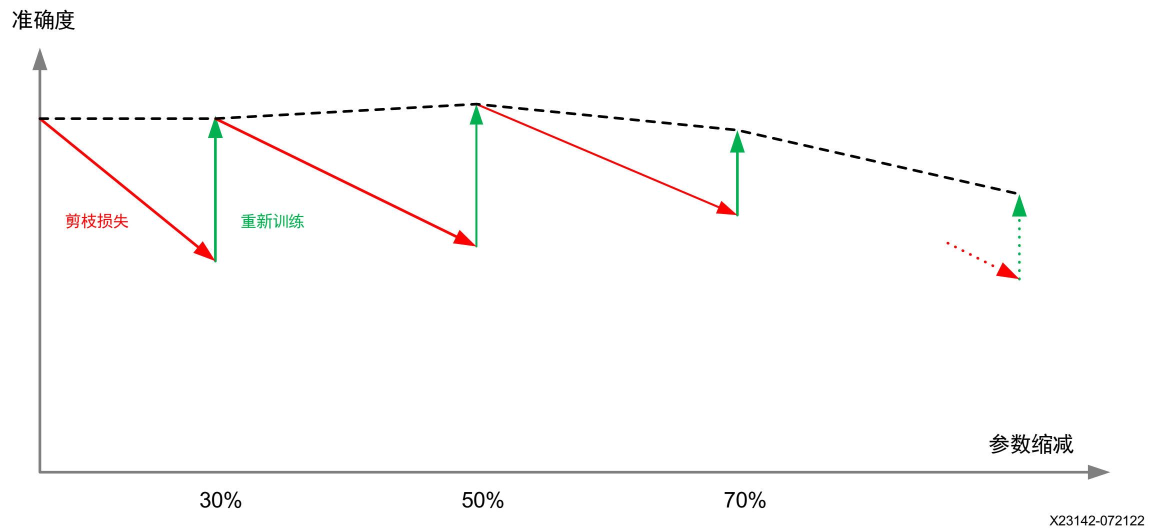
## 迭代剪枝

剪枝算法的设计使其能在减少模型参数数量的同时尽可能降低精度损失。这是一个迭代进程，如下图所示。剪枝会导致精度损失，而通过训练对剩余权重进行微调则可恢复精度。经过训练有素且未剪枝的模型充当第一次迭代的输入，称为基线模型。该模型会加以剪枝和微调。接下来，从上一次迭代得到的微调模型即成为新的基线，并再次进行剪枝和微调。此进程会反复进行多次迭代，直到达到所需的稀疏度。迭代方法是必需的，因为在单次传递中，无法在维持精度的同时对具有高剪枝率的模型进行剪枝。单次迭代中如果剪枝的参数过多，那么精度损失可能过于剧烈，导致不可能通过微调来恢复精度。

**重要提示！** 每次迭代中会逐渐减少参数以改善微调阶段的精度。

利用迭代剪枝的进程，可以达到更高的剪枝率，同时模型性能不会出现显著损失。

图 12：**迭代剪枝**



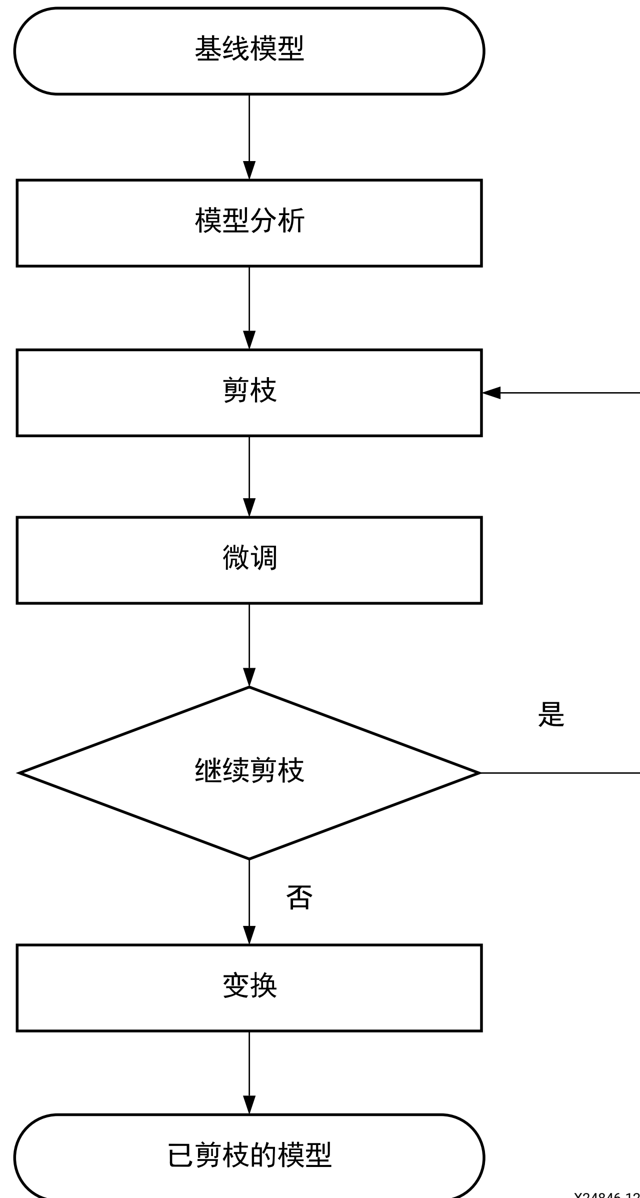

以下描述了迭代剪枝的 4 个主要阶段：

- 分析：对模型执行敏感度分析，判定最优剪枝策略。
- 剪枝：减少输入模型中的计算次数。
- 微调：重新训练已剪枝的模型以恢复精度。
- 变换：生成含更低权重的密集模型。

## 模型剪枝

请遵循上述步骤进行模型剪枝。下图同样显示了这些步骤。

图 13：**迭代剪枝工作流程**




1. 分析原始基线模型。
2. 对模型进行剪枝。
3. 对剪枝后的模型进行微调。
4. 多次重复步骤 2 和 3，直至在精度与稀疏度之间达成期望的平衡。
5. 将剪枝后的稀疏模型变换为最终密集加密的模型，以供在 Vitis AI 量化器中使用。

## 改善剪枝结果的建议

以下提供了剪枝结果最优化建议的列表。通过遵守这些准则，开发者发现剪枝率提高了，精度损失也降到了最低。

- 请尽可能使用更多数据来执行模型分析。理想情况下，应使用确认数据集中的所有数据，但这可能较为耗时。您也可以使用部分确认数据集来确保有至少一半的数据集得到了使用。
- 微调阶段中，请利用少数超参数进行实验，包括初始学习率和学习率衰减策略。使用最佳结果作为下一次迭代的输入。
- 微调中所使用的数据应为用于训练基线模型的原始数据集的子集。
- 如果多次微调实验后，精度并未能显著改善，请尝试缩减剪枝率并重新运行剪枝和微调。

## 神经架构搜索

神经架构搜索 (NAS) 的概念是指，对于任意给定推断任务和数据集，在潜在设计空间内都多个网络架构，这些架构不仅有效而且具有极高的精度得分。通常开发者从熟悉的标准主干（例如，ResNet50）开始，并对该网络进行训练以得到最佳精度。但在许多情况下，计算成本低得多的网络拓扑即可提供相似或更好的性能。对于开发者而言，使用相同数据集来训练多个网络（有时最多局限于使其成为训练超参数）并非选择最佳网络拓扑的有效方法。

NAS 可灵活应用于每个层次。通过最大程度减少剪枝后的网络损失，即可知晓通道数量或稀疏度。NAS 能在速度与精度之间成功达成平衡，但需要大量时间进行训练。此方法需执行四步式进程：

1. 训练
2. 搜索
3. 剪枝
4. 微调（可选）

相比于低精度剪枝，单样本 NAS 实现可将多个候选“子网络”汇编为单个过度参数化的计算图，称为“超网”(Supernet)。训练最优化算法会尝试使用监督学习对所有候选网络同时进行最优化。完成此训练进程后，候选子网络会基于计算成本和精度进行排名。开发者可选择满足自己要求的最适合的候选子网络。单样本 NAS 方法能有效压缩逐通道卷积模型和传统卷积模型，但需要较长的训练时间和较高的技术水平。

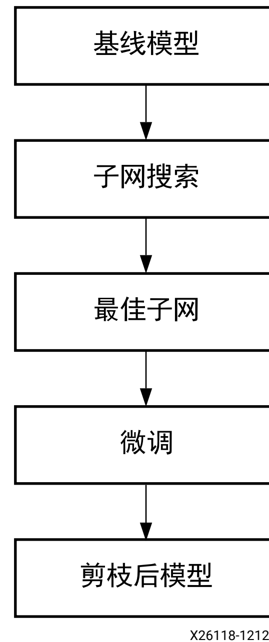
## 单步剪枝

单步剪枝会实现 EagleEye[#unique_33/unique_33_Connect_42_li_l3s_zvn_dsb] 算法。它仅通过采用了一个简单而又高效的评估组件，就得以在已剪枝模型及其对应微调精度之间引入强大的正关联，这个组件名为自适应批量归一化。它使您无需进行模型微调，即可获取潜在精度最高的子网络。简而言之，单步剪枝方法会搜索一群满足所需模型大小的子网络（即，生成的剪枝后模型），并选择其中最有潜力的子网络。所选子网络经过重新训练后即可恢复精度。

剪枝步骤如下所示：

1. 搜索满足所需剪枝率的子网络。
2. 从多个具有评估组件的子网络中选择潜在网络。
3. 对剪枝后的模型进行微调。

图 14：**单步剪枝工作流程**




**注释：**Bailin Li et al., EagleEye: Fast Sub-net Evaluation for Efficient Neural Network Pruning, arXiv:2007.02491

## Once-for-All (OFA)

Once-for-All (OFA)[1] 是基于 One-Shot NAS 的压缩方案。您经常会执行诸如在一个或多个器件上压缩和部署经过训练的模型之类的任务。传统技巧要求您为每个器件重复网络设计进程，并从头开始重新训练设计的网络，但由此会导致难以承担的计算成本。

OFA 引入了全新的解决方案来应对这一难题：即设计 Once-For-All（一次训练多次部署）网络，此网络可在多种多样的架构配置之下直接部署。故而可以分摊训练成本。只需选择 Once-For-All 网络中的一部分即可执行推断。

OFA 可以在远超剪枝的更多维度内降低模型大小。它可以构建一系列不同深度、宽度、内核大小和图像分辨率的模型，并对所有候选模型进行联合最优化。经训练后，进化搜索即可发现在精度与吞吐量之间达到最佳平衡的子网络。

对于原始模型中的每一层，Vitis OFA 允许您使用任意通道剪枝率和任意内核大小。原始模型拆分为具有共享权重的众多子网络，并成为一个超网络。子网络可以执行前向传递，并使用卷积权重的一部分进行更新。所有子网络都应在训练期间进行联合最优化。当所有子网络都妥善完成训练后，您可从超网络中搜索已达成精度与吞吐量的最佳平衡的子网络。

为了获得最高压缩率，Vitis OFA 剪枝器可以基于逐通道卷积数量与常规卷积数量的比率来最优化搜索空间。如果逐通道卷积数量超过常规卷积数量，那么 Vitis OFA 剪枝器主要对内核大小 > 1 的卷积层进行压缩。这将导致超网络中通道宽度狭窄且精度下降。

OFA 使用原始模型作为教师模型来指导子网络的训练。知识蒸馏允许将教师模型的输出用作为软化标签，以提供有关类内和类间的更多信息。Vitis OFA 使用自适应软知识蒸馏损失 (KDLoss[2]) 和三明治规则 [3] 来改善性能和效率。相比于原始 OFA，Vitis OFA 可将训练时间减半。

**注释：**

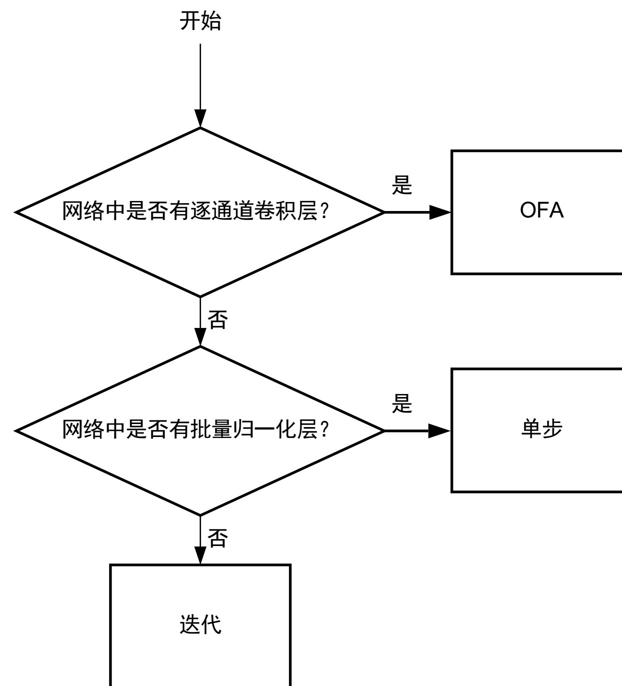
1. Han Cai et al., Once-for-All: Train One Network and Specialize it for Efficient Deployment, arXiv:1908.09791
2. Dilin Wang et al., AlphaNet: Improved Training of Supernets with Alpha-Divergence, arXiv:2102.07954

3. Jiahui Yu et al., BigNAS: Scaling Up Neural Architecture Search with Big Single-Stage Models, arXiv:2003.11142

## 剪枝方法选择

Vitis AI 优化器中为 PyTorch 提供了 3 种剪枝方法。请参阅以下决策树以选择适合您的网络的方法。

图 15：在 **PyTorch** 中选择剪枝方法的流程图




# TensorFlow (1.15) 版本 - vai_p_tensorflow

您必须在剪枝前先创建 TensorFlow 会话，此会话中包含计算图和已由 TensorFlow 初始化器、检查点、SavedModel 等完成初始化的变量。Vitis 优化器 TensorFlow 会就地执行计算图剪枝，并提供相应方法用于导出已冻结且已剪枝的计算图。

存储器中经剪枝的计算图是稀疏的，保留了原有的权重形状，同时移除特定的通道并将其设置为零。已冻结且已剪枝的计算图导出后即为大小更小的精简计算图，这意味着非必要的通道均已移除。

## 准备基线模型

这里使用了一个简单的 MNIST convnet。

```
model = keras.Sequential([
    layers.InputLayer(input_shape=input_shape),
    layers.Conv2D(16, kernel_size=(3, 3), activation="relu"),
    layers.BatchNormalization(),
```

```
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Flatten(),
    layers.Dropout(0.5),
    layers.Dense(num_classes),
])
```

## 例化 TensorFlow 会话

```
with tf.Session() as sess:
  model, input_shape = mnist_convnet()
  sess.run(tf.global_variables_initializer()) # initializing variables
```

## 创建剪枝运行器

以下提供了一个实例用于获取剪枝运行器，其中包含 TensorFlow 会话、输入规格和输出节点名称：

```
from tf1_nndct.optimization.pruning import IterativePruningRunner

with tf.Session() as sess:
  model, input_shape = mnist_convnet()
  sess.run(tf.global_variables_initializer())
  input_specs={'input_1:0': tf.TensorSpec(shape=(1, 28, 28, 1),
dtype=tf.dtypes.float32)}
  pruner = IterativePruningRunner("mnist", sess, input_specs, ["dense/
BiasAdd"])
```

## 模型剪枝

要对模型进行剪枝，请执行以下步骤：

1. 定义用于模型性能评估的函数，其类型为 `Callable[[tf.compat.v1.GraphDef], float]`。唯一实参是一个冻结的计算图，它是剪枝进程的中间结果。剪枝运行程序会执行多次剪枝，以查找最优剪枝策略。

   ```
   def eval_fn(frozen_graph_def: tf.compat.v1.GraphDef) -> float:
     with tf.compat.v1.Session().as_default() as sess:
       tf.import_graph_def(frozen_graph_def, name="")

       # do evaluation here

       return 0.5 # Returning a constant is for demonstration purpose
   ```

2. 使用此评估函数来运行模型分析：您可指定用于模型分析的器件。默认值为 ['/GPU:0']。如果给定多个器件，那么剪枝运行程序会在每个器件上运行并行模型分析。

   ```
   pruner.ana(eval_fn, gpu_ids=['/GPU:0', '/GPU:1'])
   ```

3. 确定剪枝稀疏度。此比率表示前向传递中模型的浮点计算量的缩减情况。`pruned_model's FLOPs = (1 - sparsity) * original_model's FLOPs`。此比率值应在 (0, 1) 范围内：

   ```
   shape_tensors, masks = pruner.prune(sparsity=0.5)
   ```

**注释：**`sparsity` 只是一个近似目标值，实际剪枝率与该值并不完全相同。shape_tensors 是表示 NodeDef 映射的字符串。键为 graph_def 中 node_def 的名称，需更新后方可获取精简计算图。`masks` 对应于变量。它仅包含 0 和 1。调用此方法后，会话中的计算图将进行剪枝，并变为稀疏计算图。

4. 导出冻结的精简计算图。使用 `prune` 方法所返回的 `shape_tensors` 和 `masks` 来生成冻结的精简计算图，如下所示：

```
slim_graph_def = pruner.get_slim_graph_def(shape_tensors, masks)
```

## 微调稀疏模型

计算图采用就地剪枝，因此您可像原始计算图一样对剪枝后的计算图进行微调。TensorFlow 优化器会自动应用掩码。

# vai_p_tensorflow API

**tf_nndct1.IterativePruningRunner**

```
__init__(self,
         model_name: str,
         sess: SessionInterface,
         input_specs: Mapping[str, tf.TensorSpec],
         output_node_names: List[str],
         excludes: List[str]=[])
```

实参：

- model_name： 模型名称。
- sess：TensorFlow 会话的实例包含计算图和初始化的变量。
- input_specs： 此映射的键是基线模型的输入节点名称。
- output_node_names：目标输出节点名称。
- excludes：跳过剪枝的节点名称。

返回：IterativePruningRunner 的实例

```
ana(self,
    eval_fn: Callable[[tf.compat.v1.GraphDef], float],
    gpu_ids: List[str]=['/GPU:0'],
    checkpoint_interval: int = 10) -> None:
```

实参

- eval_fn：此函数用于评估剪枝进程的中间结果。需返回浮点。
- gpu_ids：该字符串列表表示要运行评估的器件。
- checkpoint_interval：此方法会实现高速缓存机制，并保存每次执行 checkpoint_interval 评估的结果。

返回：无

```
prune(self,
      sparsity: float=None,
      threshold: float=None,
      max_attemp: int=10) -> Tuple[Mapping[str, TensorProto], Mapping[str,
np.ndarray]]:
```

有两种剪枝模式：基于 FLOP 的方法和基于精度的方法，对应实参稀疏度和阈值。这两个实参不得同时设为 None。实参稀疏度的优先级高于阈值。

实参：

- sparsity：此比率表示前向传递中模型的浮点计算量的缩减情况。
- threshold：范围为 [0, 1]。表示剪枝后的计算图与原始计算图之间最大可接受的相对差值。
- max_attemp：剪枝运行程序会以迭代方式查找最优剪枝策略，执行 max_attemp 个步骤后无论如何都会返回结果。

返回：

- shape_tensors：该字符串表示 NodeDef 映射。键为 graph_def 中 node_def 的名称，需更新后方可获取精简计算图。值为目标 node_def 内容掩码。
- masks：表示对应于变量的字符串到阵列映射。

```
get_slim_graph_def(self,
                   shape_tensors: Mapping[str, TensorProto]=None,
                   masks: Mapping[str, np.ndarray]=None) ->
tf.compat.v1.GraphDef:
```

实参：

- shape_tensors：从 prune 方法返回的字符串到 NodeDef 映射。
- masks：表示对应于变量的字符串到阵列映射。该对象是从 prune 方法获取的。

返回：冻结的精简 graph_def。

## TensorFlow 示例

请参阅 GitHub 上的 Vitis AI 仓库。

# TensorFlow (2.x) 版本 - vai_p_tensorflow2

vai_p_tensorflow2 仅支持由函数式 API 或 Sequential API 所创建的 Keras 模型。不支持子类化模型。

## 创建模型

此处使用了来自 Keras vision 示例的一个简单的 MNIST convnet。

```
model = keras.Sequential([
    keras.Input(shape=input_shape),
    layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Flatten(), layers.Dropout(0.5),
    layers.Dense(num_classes, activation="softmax"),
])
```

## 创建剪枝运行器

要创建包含 shape 和 dtype 信息的输入规格，并随后将其用于获取剪枝运行器，请使用以下命令：

```
from tf_nndct.pruning import IterativePruningRunner

input_shape = [28, 28, 1]
input_spec = tf.TensorSpec((1, *input_shape), tf.float32)
runner = IterativePruningRunner(model, input_spec)
```

## 模型剪枝

要对模型进行剪枝，请执行以下步骤：

1. 定义用于模型性能评估的函数。该函数必须满足以下两项要求：
   - 第一个实参必须是要评估的 `keras.Model` 实例。
   - 它必须返回 Python 值，以指示模型性能。

   ```
   def evaluate(model):
     model.compile(loss="categorical_crossentropy", optimizer="adam",
   metrics=["accuracy"])
     score = model.evaluate(x_test, y_test, verbose=0)
     return score[1]
   ```

2. 请使用此评估函数来运行模型分析：

   ```
   runner.ana(evaluate)
   ```

3. 确定剪枝率。此比率表示前向传递中模型的浮点计算量的缩减情况。

   [剪枝后模型的 MAC 数] = (1 – 比率) * [原始模型的 MAC 数]

   该比率值应介于 0 到 1 之间：

   ```
   sparse_model = runner.prune(ratio=0.2)
   ```

   **注释：** `ratio` 只是一个近似目标值，实际剪枝率与该值并不完全相同。

从 `prune()` 返回的模型是稀疏模型，这表示剪枝后的通道权重设为零 (0)，并且模型大小保持不变。此稀疏模型将在下一轮迭代剪枝中使用。仅限完成剪枝后，此稀疏模型才会转换为剪枝后的密集模型。

除了返回稀疏模型外，剪枝运行器还会在 `.vai` 目录中生成规范文件，用于描述每个层采用的剪枝方式。

## 微调稀疏模型

训练稀疏模型与训练标准模型并无区别。除调整超参数外，无需执行任何其他操作。

```
sparse_model.compile(loss="categorical_crossentropy", optimizer="adam",
metrics=["accuracy"])
sparse_model.fit(x_train, y_train, batch_size=128, epochs=15,
validation_split=0.1)
sparse_model.save_weights("model_sparse_0.2", save_format="tf")
```

**注释：**调用 `save_weights` 时，请使用“tf”格式来保存权重。

## 执行迭代剪枝

从先前微调阶段保存的检查点执行加载。增大剪枝率值，提升稀疏度等级。执行每个剪枝步骤后，对此稀疏模型进行微调。重复此剪枝和微调循环，直至稀疏度达到目标值，或者直至观察到评估性能等级显著降级为止。

```
model.load_weights("model_sparse_0.2")

input_shape = [28, 28, 1]
input_spec = tf.TensorSpec((1, *input_shape), tf.float32)
runner = IterativePruningRunner(model, input_spec)
sparse_model = runner.prune(ratio=0.5)
```

## 获取剪枝后的模型

完成迭代剪枝后，即可生成稀疏模型，此模型所含参数数量与原始模型相同，但其中大量参数现已设置为零 (0)。

调用 `get_slim_model()` 移除稀疏模型中含 0 值的参数并生成最终剪枝后的模型：

```
model.load_weights("model_sparse_0.5")

input_shape = [28, 28, 1]
input_spec = tf.TensorSpec((1, *input_shape), tf.float32)
runner = IterativePruningRunner(model, input_spec)
slim_model = runner.get_slim_model()
```

默认情况下，运行器使用最新剪枝规范来生成精简模型。您可以看到，最新规范文件含如下命令：

```
$ cat .vai/latest_spec
$ ".vai/mnist_ratio_0.5.spec"
```

如果此文件与您的稀疏模型不匹配，您可显式指定要使用的文件路径：

```
slim_model = runner.get_slim_model(".vai/mnist_ratio_0.5.spec")
```

您可使用 Keras 模型保存 API 保存精简模型并重新加载此模型用于推断或量化。例如：

```
slim_model.save('/tmp/model')
loaded_model = tf.keras.models.load_model('/tmp/model')
```

## vai_p_tensorflow2 API

### tf_nndct.IterativePruningRunner

此运行器适用于以迭代方式对模型进行结构化剪枝。此 API 具有下列方法：

- ```
  __init__(model, input_specs)
  ```

  创建新的 `IterativePruningRunner` 对象。

  - model：要进行剪枝的基线模型。模型应为 `keras.Model` 的实例。
  - input_specs：单一或列表形式的 `tf.TensorSpec`，用于表示模型输入规范。

- ```
  ana(eval_fn, excludes=None, forced=False)
  ```

  执行模型分析。分析结果保存在 `'.vai'` 目录中，除非 `forced` 设为 True，否则此缓存结果将在后续调用中直接使用。

  - eval_fn：可调用对象，取 `keras.Model` 对象作为其第一个实参，并返回评估得分。
  - excludes：要从剪枝中排除的层名称或层实例的列表。
  - forced：此项设为 True 时，会运行模型分析来替代缓存的分析结果。

- ```
  prune(ratio=None, threshold=None, spec_path=None, excludes=None,
  mode='sparse')
  ```

  对基线模型进行剪枝，并返回稀疏模型。剪枝程度可通过三种方式来指定：比率、阈值或剪枝规范。首选第一种方法；后两种方法更适合搭配手动微调来进行实验。

  - ratio：期望的基线模型 FLOP（每秒浮点运算次数）缩减百分比。这是指导值，实际 FLOP 缩减与该值并非严格相等。
  - threshold：基线模型与剪枝后的模型之间的相对模型性能损失比例。
  - spec_path：用于模型剪枝的剪枝规范路径。
  - excludes：要从剪枝中排除的层名称或层实例的列表。
  - mode：基线模型的剪枝模式，剪枝后返回稀疏模型。

- ```
  get_slim_model(spec_path=None)
  ```

  从稀疏模型获取精简模型。默认使用最新剪枝规范来执行此变换。如果稀疏模型不是根据最新规范生成的，则可显式提供规范路径。

  - spec_path：剪枝规范路径会将稀疏模型变换为精简模型。

## TensorFlow2 示例

请参阅 GitHub 上的 Vitis AI 仓库。

# PyTorch 版本 - vai_p_pytorch

PyTorch 上的剪枝工具是 Python 程序包，而不是可执行程序，vai_p_pytorch 可提供三种模型剪枝方法：

- 迭代剪枝
- 单步剪枝
- Once-for-All (OFA)

迭代剪枝和单步剪枝适合具有传统卷积层的网络，但对于基于逐通道卷积的网络（如，MobileNet-v2），效果并不好。卷积神经网络 (CNN) 通常包含 BatchNormalization（批量归一化）层，对于这些网络，首选单步剪枝，因为这种方法更快且更有效。如果诸如 VGGNet 之类的网络中没有 BatchNormalization 层，那么应使用迭代剪枝。

OFA 对于逐通道卷积和传统卷积都适用。重要的是明确 OFA 理论上是这三种方法中最好的方法，但不一定能轻松获得理想的剪枝结果。结果取决于超网络的训练优度，并且可能需要较长的训练时间和强大的训练技能。

总结，如果网络中有 BatchNormalization 层，请使用单步剪枝。否则，使用迭代剪枝。如果对剪枝结果不满意，那么 OFA 可用作为替代方法。

# 高精度剪枝

## 创建模型

为简便起见，此处使用来自 `torchvision` 的 ResNet50。

```
from torchvision.models.resnet import resnet50
model = resnet50(pretrained=True)
```

## 创建稀疏剪枝器

剪枝器需要两个实参：

- 要剪枝的模型
- 模型推断所需的输入

```
import torch
from pytorch_nndct import SparsePruner

inputs = torch.randn([1, 3, 224, 224], dtype=torch.float32)
pruner = SparsePruner(model, inputs)
```

## 生成稀疏模型

调用 `sparse_model()` 以获取稀疏模型。此方法会查找所有 `nn.Conv2d` / `nn.ConvTranspose2d` 和 `nn.BatchNorm2d` 模块，并将这些模块替换为 `DynamicConv2d` / `DynamicConvTranspose2d` 和 `DynamicBatchNorm2d`。此方法会将满足稀疏度条件的 `nn.Conv2d` / `nn. linear` 层替换为 `SparseConv2d` / `SparseLinear`。

此方法支持 `nn.Conv2d` / `nn. linear` 权重和激活同时执行剪枝。激活的稀疏度可为 0 或 0.5。当激活的稀疏度为 0 时，权重的稀疏度可设为 0、0.5 或 0.75。当激活的稀疏度为 0.5 时，权重的稀疏度只能设为 0.75。block_size 是输入通道的连续元素数。通道根据权重/激活展开。它设为 4、8 或 16。因此，如果卷积的输入通道权重大于 16，就会被替换为稀疏卷积。

```
sparse_model =
sparse_pruner.sparse_model(w_sparsity=0.5,a_sparsity=0,block_size=4)
```

重新训练稀疏模型与创建基线模型并无不同。知识蒸馏可以实现更好的精度。

## 获取稀疏模型

调用 `export_sparse_model()` 以获取从稀疏模型转换所得网络及稀疏权重，用于在硬件上为指定稀疏计算执行推断。

```
model = sparse_pruner.export_sparse_model(sparse_model)
```

# 低精度剪枝

根据此方法对模型进行剪枝的步骤如下所示：

## 创建模型

为简便起见，此处使用来自 `torchvision` 的 ResNet18。

```
from torchvision.models.resnet import resnet18
model = resnet18(pretrained=True)
```

## 创建剪枝运行器

导入模块并准备输入特征符：

```
from pytorch_nndct import get_pruning_runner

# The input signature should have the same shape and dtype as the model
input.
input_signature = torch.randn([1, 3, 224, 224], dtype=torch.float32)
```

创建迭代剪枝运行器：

```
runner = get_pruning_runner(model, input_signature, 'iterative')
```

或者单步剪枝运行器：

```
runner = get_pruning_runner(model, input_signature, 'one_step')
```

## 模型剪枝

### 迭代剪枝

此方法包含两个阶段：模型分析和剪枝后的模型生成。完成模型分析后，分析结果会保存在名为 `.vai/xxx.sens` 的文件内。您可使用此文件对模型进行迭代剪枝。在迭代剪枝中，需逐渐对模型进行剪枝以达到目标稀疏度。这是通过使用迭代循环来达成的，此循环由剪枝步骤和微调步骤组成，并按步骤使用适度的剪枝率。如果尝试设置过高的剪枝率，则会导致步骤中精度损失且无法恢复。

1. 定义评估函数。该函数必须取一个模型作为其首个实参，并返回得分。

```
def eval_fn(model, dataloader):
  top1 = AverageMeter('Acc@1', ':6.2f')
  model.eval()
  with torch.no_grad():
  for i, (images, targets) in enumerate(dataloader):
    images = images.cuda()
    targets = targets.cuda()
    outputs = model(images)
    acc1, _ = accuracy(outputs, targets, topk=(1, 5))
    top1.update(acc1[0], images.size(0))
  return top1.avg
```

2. 运行模型分析并获取剪枝后的模型。

```
runner.ana(eval_fn, args=(val_loader,))

model = pruning_runner.prune(removal_ratio=0.2)
```

模型分析只需执行一次即可。您可对模型进行迭代剪枝，而无需重新运行分析，因为针对特定剪枝率仅生成一个剪枝后的模型。通过剪枝获取的子网络一般都不够好，因为近似算法会根据分析结果来生成此独特剪枝模型。单步剪枝方法可以生成更好的子网络。

**单步剪枝**

此方法包含两个阶段：基于自适应 BN 搜索剪枝策略以及剪枝后的模型生成。搜索后，会生成名为 `.vai/xxx.search` 的文件，用于存储搜索结果（剪枝策略和对应的评估得分）。您可单步获取最终剪枝后模型。

`num_subnet` 可提供满足稀疏性要求（待确认）的候选子网络的目标数量。您可从这些候选结果中选择最佳子网络。该值越高，搜索耗时越长，但找到更好的子网络的可能性更高。

```
# Adaptive-BN-based searching for pruning strategy. 'calibration_fn' is a function for calibrating BN layer's statistics.
runner.search(gpus=['0'], calibration_fn=calibration_fn,
calib_args=(val_loader,), eval_fn=eval_fn, eval_args=(val_loader,),
num_subnet=1000, removal_ratio=0.7)

model = runner.prune(removal_ratio=0.7, index=None)
```

`eval_fn` 与迭代剪枝方法相同。以下代码示例中显示了用于实现自适应 BN 的 `calibration_fn` 函数。您应采用类似方式来定义自己的代码。

```
def calibration_fn(model, dataloader, number_forward=100):
  model.train()
  with torch.no_grad():
    for index, (images, target) in enumerate(dataloader):
      images = images.cuda()
      model(images)
    if index > number_forward:
      break
```

相比于迭代方法，单步剪枝方法具有几项优势：

- 生成的剪枝后模型通常更准确。满足要求的所有子网络都会进行评估。
- 工作流程更为简单，因为您可单步获取最终剪枝后的模型，无需迭代。
- 重新训练精简模型比重新训练稀疏模型更快。

单步剪枝有两大劣势：其一是随机生成的剪枝策略不可预测。另一项是必须对每个剪枝比率执行一次子网络搜索。

## 重新训练模型

重新训练模型与创建基线模型并无不同。

```
optimizer = torch.optim.Adam(model.parameters(), 1e-3, weight_decay=5e-4)
best_acc1 = 0

for epoch in range(args.epoches):
  train(train_loader, model, criterion, optimizer, epoch)
  acc1, acc5 = evaluate(val_loader, model, criterion)

  is_best = acc1 > best_acc1
  best_acc1 = max(acc1, best_acc1)
  if is_best:
    torch.save(model.state_dict(), 'model_pruned.pth')
    # Sparse model has one more special method in iterative pruning.
    if hasattr(model, 'slim_state_dict'):
      torch.save(model.slim_state_dict(), 'model_slim.pth')
```

## 生成经剪枝的模型

在稀疏模型中会移除剪枝后的模型中设为 0 的参数。有两种方法可用于生成最终剪枝模型。

### 使用剪枝 API

```
method = 'iterative' # or 'one_step'
runner = get_pruning_runner(model, input_signature, method)
slim_model = runner.prune(removal_ratio=0.2, mode='slim')
slim_model.load_state_dict(torch.load('model_slim.pth'))
```

### 不使用剪枝 API

此方法常用于对已剪枝的模型进行量化，因为有时无法调用剪枝 API。

```
from pytorch_nndct.utils import slim

model = create_your_baseline_model()
slim_model = slim.load_state_dict(model, torch.load('model_slim.pth'))
```

# Once-for-All (OFA)

OFA 方法的步骤如下所述：

## 创建模型

为简便起见，此处使用来自 `torchvision` 的 mobilenet_v2。

```
from torchvision.models.mobilenet import mobilenet_v2
model = mobilenet_v2(pretrained=True)
```

## 创建 OFA 剪枝器

剪枝器需要两个实参：

- 要剪枝的模型
- 模型推断所需的输入

```
import torch
from pytorch_nndct import OFAPruner

inputs = torch.randn([1, 3, 224, 224], dtype=torch.float32)
pruner = OFAPruner(model, inputs)
```

**注释：**输入无需采用真实数据。您可使用随机生成的虚拟数据，只要其形状和类型与真实数据相同即可。

## 生成 OFA 模型

调用 `ofa_model()` 以获取 OFA 模型。此方法会查找所有 `nn.Conv2d` / `nn.ConvTranspose2d` 和 `nn.BatchNorm2d` 模块，并将这些模块替换为 `DynamicConv2d` / `DynamicConvTranspose2d` 和 `DynamicBatchNorm2d`。

在最终 OFA 模型中，需包含剪枝率列表以便为每个层指定最大和最小通道剪枝率。

OFA 模型中每个卷积层的输出通道均可使用任意的剪枝率。此列表中的最大值和最小值分别表示该模型的最大和最小压缩率。列表中的其他值则表示要最优化的子网络。剪枝率默认设为 [0.5, 0.75, 1]。

对于从 OFA 模型中采样的子网络，卷积层的输出通道是剪枝率列表中的数字之一乘以其原始数字，例如，对于剪枝率列表 [0.5, 0.75, 1] 和卷积层 `nn.Conv2d(16, 32, 5)`，该层在采样子网络中的输出通道是 [0.5*32, 0.75*32, 1*32] 之一。

由于第一层和最后一层对网络性能存在显著影响，因此通常排除在剪枝之外。默认情况下，此方法会自动识别第一个卷积和最后一个卷积，并将其置于排除列表中。将 `auto_add_excludes` 设置为 False 即可取消此功能特性。

```
ofa_model = ofa_pruner.ofa_model([0.5, 0.75, 1], excludes = None,
auto_add_excludes=True)
```

## 训练 OFA 模型

此方法使用三明治规则来对所有 OFA 子网络进行联合最优化。`sample_random_subnet()` 函数用于获取子网络。动态子网络可在正向传播和反向传播中使用。

在每个训练步骤中，提供小批次数据的情况下，三明治规则会对一个“max”（最大）子网络、一个“min”（最小）子网络和两个随机子网络进行采样。每个子网络都会利用给定数据执行独立的前向/后向传递，并且所有子网络都会一起更新其参数。

```
# using sandwich rule and sampling subnet.
for i, (images, target) in enumerate(train_loader):

  images = images.cuda(non_blocking=True)
  target = target.cuda(non_blocking=True)

  # total subnets to be sampled
  optimizer.zero_grad()

  teacher_model.train()
```

```
  with torch.no_grad():
    soft_logits = teacher_model(images).detach()

  for arch_id in range(4):
    if arch_id == 0:
      model, _ = ofa_pruner.sample_subnet(ofa_model,'max')
    elif arch_id == 1:
      model, _ = ofa_pruner.sample_subnet(ofa_model,'min')
    else:
      model, _ = ofa_pruner.sample_subnet(ofa_model,'random')

    output = model(images)

    loss = kd_loss(output, soft_logits) + cross_entropy_loss(output,
target)
    loss.backward()

  torch.nn.utils.clip_grad_value_(ofa_model.parameters(), 1.0)
  optimizer.step()
  lr_scheduler.step()
```

## 搜索已约束的子网络

训练完成后，您可基于神经网络孪生来开展进化搜索，以 MAC 最小值和最大值范围来获取已达成 MAC 与准确度的最佳平衡的子网络。

```
pareto_global = ofa_pruner.run_evolutionary_search(ofa_model,
calibration_fn, (train_loader,) eval_fn, (val_loader,), 'acc1', 'max',
min_macs=230, max_macs=250)

ofa_pruner.save_subnet_config(pareto_global, 'pareto_global.txt')
```

进化搜索结果如下所示：

```
{
"230": {
     "net_id": "net_evo_0_crossover_0",
     "mode": "evaluate",
     "acc1": 69.04999542236328,
     "macs": 228.356192,
     "params": 3.096728,
     "subnet_setting": [...]
}
"240": {
     "net_id": "net_evo_0_mutate_1",
     "mode": "evaluate",
     "acc1": 69.22000122070312,
     "macs": 243.804128,
     "params": 3.114,
     "subnet_setting": [...]
}}
```

## 获取子网络

调用 `get_static_subnet()` 以获取特定子网络。static_subnet 可用于微调和量化。

```
pareto_global = ofa_pruner.load_subnet_config('pareto_global.txt')
static_subnet, static_subnet_config, flops, params = \
ofa_pruner.get_static_subnet(ofa_model, pareto_global['240']
['subnet_setting'])
```

# vai_p_pytorch API

## pytorch_nndct.SparsePruner

此 API 具有下列方法：

- `__init__(model, inputs)`
  - model：要进行剪枝的 `torch.nn.Module` 对象。
  - inputs：单个 torch 或 torch 列表。张量用作为模型推断的输入。输入无需采用真实数据。可以采用随机生成的张量，与真实数据的形状和数据类型相同即可。
- `sparse_model(w_sparsity=0.5, a_sparsity=0, block_size=16,excludes=None)`
  - w_sparsity：下列值之一：['0', '0.5', '0.75']。卷积层和完全连接的层的权重稀疏度浮点值。默认 w_sparsity 设为 0.5。
  - a_sparsity：下列值之一：['0', '0.5']。激活的稀疏度浮点值。此处激活表示稀疏层的输入特征映射。默认 a_sparsity 设为 0。如果 a_sparsity 为 0.5，那么 w_sparsity 必须为 0.75。
  - block_size：输入通道或根据权重/激活展开的通道的连续元素数 (int)。
  - excludes：要从稀疏剪枝中排除的模块列表。
- `export_sparse_model(model)`

  返回从稀疏模型转换所得网络及稀疏权重，用于在硬件上为指定稀疏计算执行推断。

  - model：稀疏模型。

## pytorch_nndct.get_pruning_runner

此 API 具有下列方法：

```
get_pruning_runner(model, inputs, method)
```

- model：要进行剪枝的 `torch.nn.Module` 对象。
- inputs：单个 torch 或 torch 列表。张量用作为模型推断的输入。它无需采用真实数据。可以采用随机生成的张量，与真实数据的形状和数据类型相同即可。
- method：可取“iterative”或“one_step”。

## pytorch_nndct.IterativePruningRunner

此 API 具有下列方法：

- `__init__(model, inputs)`
    - model：要进行剪枝的 `torch.nn.Module` 对象。
    - inputs：单个 torch 或 torch 列表。张量用作为模型推断的输入。输入无需采用真实数据。可以采用随机生成的张量，与真实数据的形状和数据类型相同即可。
- `ana(eval_fn, args=(), gpus=None, excludes=None, forced=False)`
    - eval_fn：可调用对象，取 `torch.nn.Module` 对象作为其第一个实参，并返回评估得分。
    - args：传递给 eval_fn 的实参元组。
    - gpus：要使用的 GPU 索引的元组或列表。如不设置，则使用默认 GPU。
    - excludes：要从剪枝中排除的节点名称或 torch 模块的列表。
    - forced：如为 False，则跳过模型分析并使用缓存的结果。
- `prune(removal_ratio=None, threshold=None, spec_path=None, excludes=None, mode='sparse')`
    - removal_ratio：期望的 MAC 缩减百分比。
    - threshold：可承受的模型性能损失的相对比例。
    - spec_path：预定义的剪枝规范。
    - excludes：要从剪枝中排除的节点名称或 torch 模块的列表。
    - mode：以下值之一：['sparse', 'slim']。在迭代循环中请始终使用 'sparse'。slim（精简）模型用于量化感知训练。

## pytorch_nndct.OneStepPruningRunner

此 API 具有下列方法：

- `__init__(model, inputs)`
    - model：要进行剪枝的 `torch.nn.Module` 对象。
    - inputs：单个 torch 或 torch 列表。张量用作为模型推断的输入。输入无需采用真实数据。可以采用随机生成的张量，与真实数据的形状和数据类型相同即可。
- `search(gpus=['0'], calibration_fn=None, calib_args=(), num_subnet=10, removal_ratio=0.5, excludes=[], eval_fn=None, eval_args=())`
    - gpus：要使用的 GPU 索引的元组或列表。如不设置，则使用默认 GPU。
    - calibration_fn：可调用对象，取 `torch.nn.Module` 对象作为其首个实参。它用于为 BatchNormalization 层校准统计数据。
    - calib_args：传递给 calibration_fn 的实参元组。
    - num_subnet：满足 MAC 约束的子网络数量。
    - removal_ratio：期望的 MAC 缩减百分比。
    - excludes：需从剪枝中排除的模块。
    - eval_fn：可调用对象，取 `torch.nn.Module` 对象作为其第一个实参，并返回评估得分。
    - eval_args：传递给 eval_fn 的实参元组。

- `prune(mode='slim', index=None, removal_ratio=None, pruning_info_path=None)`
    - mode：以下值之一：['sparse', 'slim']。对于单步方法，应始终使用 'slim'（精简）模式。
    - index：子网络索引。默认会自动选中最优子网络。
    - removal_ratio：期望的 MAC 缩减百分比。
    - pruning_info_path：JSON 文件。为当前模型保存详细的剪枝信息。可生成含该文件和原始模型的精简模型。

## pytorch_nndct.OFAPruner

此 API 具有下列方法：

- `__init__(model, inputs)`
    - model：要进行剪枝的 `torch.nn.Module` 对象。
    - inputs：单个 torch 或 torch 列表。张量用作为模型推断的输入。输入无需采用真实数据。可以采用随机生成的张量，与真实数据的形状和数据类型相同即可。
- `ofa_model(expand_ratio, channel_divisble=8, excludes=None, auto_add_excludes=True, save_search_space=False)`
    - expand_ratio：每个卷积层的剪枝率列表。OFA 模型中每个卷积层的输出通道均可使用任意的剪枝率。

      此列表中的最大值和最小值分别表示该模型的最大和最小压缩率。其他值则表示要最优化的子网络。剪枝率默认设为 [0.5, 0.75, 0.1]。
    - channel_divisible：可除以给定除数的通道数量。
    - excludes：要从剪枝中排除的模块列表。
    - auto_add_excludes：布尔值。如果该值为 True，那么此方法会自动识别第一个卷积和最后一个卷积，并将其置于排除列表中。如果为 False，则跳过创建排除列表。默认值为 True。
    - save_search_space：布尔值。如果该值为 True，则将模型的搜索空间保存为“searchspace.config”文件。您可以检查每个层的搜索空间。默认值为 False。
- `sample_subnet(model, mode)`

  返回子网络及其给定模式的配置。该子网络可以使用来自 OFA 模型及其设置的部分权重执行前向/后向传递进程。
    - model：OFA 模型。
    - mode：下列值之一：['random', 'max', 'min']。
- `reset_bn_running_stats_for_calibration(model)`

  复位 Batch Normalization 层的运行统计数据。
    - model：OFA 模型。
- `run_evolutionary_search(model, calibration_fn, calib_args, eval_fn, eval_args, evaluation_metric, min_or_max_metric, min_macs, max_macs, macs_step=10, parent_popu_size=16, iteration=10, mutate_size=8, mutate_prob=0.2, crossover_size=4)`

  运行进化搜索，查找最佳子网络，该子网络的 MAC 在给定范围内。
    - model：OFA 模型。

  - calibration_fn：BatchNormalization 校准函数。所有子网络共享 OFA 模型中的权重，但在训练 OFA 模型时不存储批量归一化统计数据（平均值和方差）。训练完成后，必须使用训练数据为用于评估的每个已采样的子网络重新校准批量归一化统计数据。
  - calib_args：calibration_fn 的实参。
  - eval_fn：用于评估模型的函数。
  - eval_args：eval_fn 的实参。
  - evaluation_metric：用于记录结果的 evaluation_metric 字符串。
  - min_or_max_metric：下列值之一：['max', 'min']。要记录在进化搜索中的评估指标的最大值或最小值。例如，如果评估指标精度为 top1，则在进化搜索中记录每次迭代的最大值。但如果评估指标为平均平方误差 (mse) 或平均绝对误差 (mae)，则记录最小值。
  - min_macs：已搜索的子网络的最小 MAC 数。
  - max_macs：已搜索的子网络的最大 MAC 数。
  - macs_step：搜索 MAC 的步骤。将间隔 [min_macs, max macs] 除以 macs_step 即可对其进行分割。对于每个分割段，搜索已达成 MAC 数与精度的最佳平衡的子网络。
  - parent_popu_size：对于所含 MAC 数在给定范围内的随机子网络，此项表示对其进行采样时的初始父填充数。该数值越大，搜索时间越长，并且获取最佳结果的可能性越高。
  - iteration：搜索的迭代次数或者整个算法的周期数。
  - mutate_size：突变的大小。子网络设置的每个值都被替换为候选值列表的另一个值（概率为 mutate_prob）。
  - mutate_prob：突变的概率。
  - crossover_size：交叉的大小。对两项子网络设置进行采样并对这两项子网络设置中的任意值进行随机交换。

- ```
  save_subnet_config(setting_config, file_name)
  ```

  利用 JSON 保存动态/静态子网络设置。

  - setting_config：动态子网络设置的配置。
  - file_name：用于保存子网络设置的文件路径。

- ```
  load_subnet_config(file_name)
  ```

  - file_name：用于加载子网络设置的文件路径。

## PyTorch 示例

请参阅 GitHub 上的 Vitis AI 优化器示例。

第 3 章

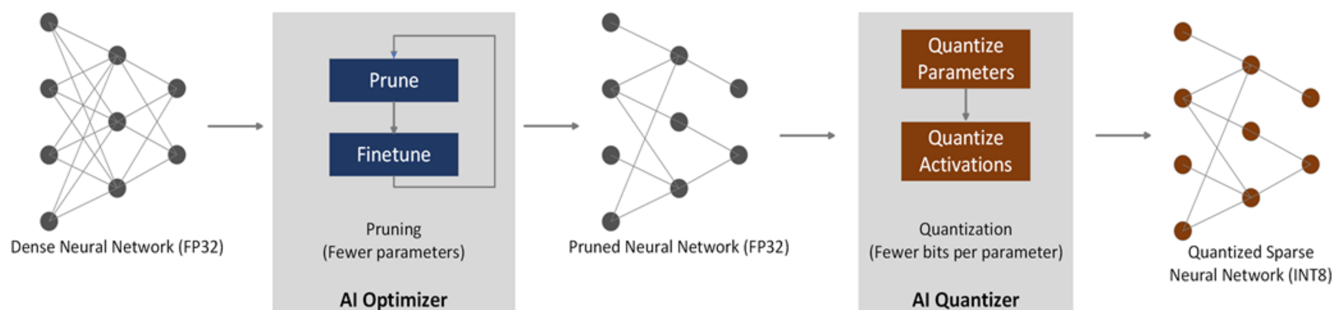
# 量化模型

## 概述

推断是一个计算密集型进程，需要大量存储器带宽来满足边缘 (Edge) 应用的低时延和高吞吐量要求。

量化和通道剪枝技巧不仅可以应对这些挑战，同时也能实现最优性能和高能效，并最大程度减少精度劣化。量化能使整数计算单元变得行之有效，并且权重与激活能以更低的精度来呈现。另一方面，剪枝可以减少所需的运算总量。AMD Vitis AI 量化器包含量化工具，而剪枝工具则集成在 Vitis AI 优化器内。

图 16：**剪枝和量化流程**




通常，进行神经网络训练时，使用的是 32 位浮点权重和激活值。 Vitis AI 量化器可将 32 位浮点权重和激活转换为 8 位整数 (INT8) 格式，这样即可降低计算复杂性，而不会损失预测精度。定点网络模型的部署所需的存储器带宽更少，因此相比浮点模型，速度更快且能效更高。Vitis AI 量化器支持神经网络中存在公用层，包括但不限于卷积、池化、完全连接和批量归一。

Vitis AI 量化器支持 TensorFlow（1.x 和 2.x）以及 PyTorch。量化器名称分别为 vai_q_tensorflow 和 vai_q_pytorch。在 Vitis AI 2.5 和更低版本中，适用于 TensorFlow 1.x 的 Vitis AI 量化器是基于 TensorFlow 1.15 来运作的，并且包含在 TensorFlow 1.15 程序包内一起发布。但从 Vitis AI 3.0 起，Vitis AI 量化器作为独立 Python 包来提供，其中特别提供了多个同时适用于 TensorFlow 1.x 和 TensorFlow 2.x 的量化 API。您导入此包后，Vitis AI 量化器即可以 TensorFlow 插件的方式来运作。

表 5：**Vitis AI 量化器支持的框架和功能**

| 模型 | 版本 | 功能特性 | | | |
|---|---|---|---|---|---|
| | | 训练后量化 (PTQ) | 量化感知训练 (QAT) | 快速微调（高级校准） | 检查器 |
| TensorFlow 1.x | 支持 1.15 | 支持 | 支持 | 支持 | 不支持 |
| TensorFlow 2.x | 支持 2.3 - 2.12 | 支持 | 支持 | 支持 | 支持 |

表 5：**Vitis AI 量化器支持的框架和功能** (续)

| 模型 | 版本 | 功能特性 | | | |
|---|---|---|---|---|---|
| | | 训练后量化 (PTQ) | 量化感知训练 (QAT) | 快速微调（高级校准） | 检查器 |
| PyTorch | 支持 1.2 到 1.13 和 2.0 | 支持 | 支持 | 支持 | 支持 |

训练后量化 (PTQ) 只需少量未标记的图像便可分析激活分布。训练后量化的运行时间因神经网络大小而异，少则数秒，多则数分钟。通常量化后，精度会有所下降，但在可接受范围内。然而，精度损失对于某些网络（如 Mobilenet）而言可能过大，甚至是难以承受的。在此类情况下，量化感知训练 (QAT) 可以进一步改善量化模型的精度。要执行 QAT，就需要原始训练数据集。此进程需要多轮微调，微调持续时间从几分钟到几小时不等。执行 QAT 时，建议使用较小的学习率。

**注释：**从 Vitis AI 1.4 起，“训练后量化”一词取代了“量化校准”一词，“量化感知训练”则取代了“量化微调”。

**注释：**Vitis AI 仅执行有符号量化。强烈建议应用标准化，其中包含通过缩放输入像素值来得到零均值和单位方差。它能确保 DPU 看到的值在 [-1.0, +1.0] 范围内。使用缩放后的无符号输入（将原始输入除以 255.0 以获得 [0.0, 1.0] 的范围）会导致动态范围丢失，因为输入范围仅用了一半。TensorFlow 2.x 和 PyTorch 量化器会提供配置来执行无符号量化用于实验。目前，对于 DPU 而言，由此获得的结果尚无法部署。

**注释：**使用 Netron 之类的工具查看模型时，对于某些层而言，存在一个 fix_point 参数，它表示用于该层的量化参数。fix_point 参数表示所使用的小数位数。例如，如果 8 位有符号量化含 fix_point = 7，那么 Q-format 表示法会显示 Q0.7，即，1 个符号位、0 个整数位和 7 个小数位。要将 Q-format 的整数值转换为浮点，请将该整数值乘以 2^-fixed_point。

对于训练后量化，会实现跨层均衡算法 [1]。跨层均衡可以提升校准性能，对于包含逐通道卷积的网络尤其如此。

通过少量未标记的数据，AdaQuant 算法 [2] 不仅可以对激活进行校准，还可以对权重进行微调。AdaQuant 使用少量未标记的数据（与训练后量化相似），但它会改变模型，这与微调类似。Vitis AI 量化器可实现此算法，并将其称为“快速微调”或“高级校准”。相比于训练后量化，快速微调能达成更高性能，但速度稍慢。

**注释：**对于快速微调，每一轮运行都会得到不同结果。这与微调类似。

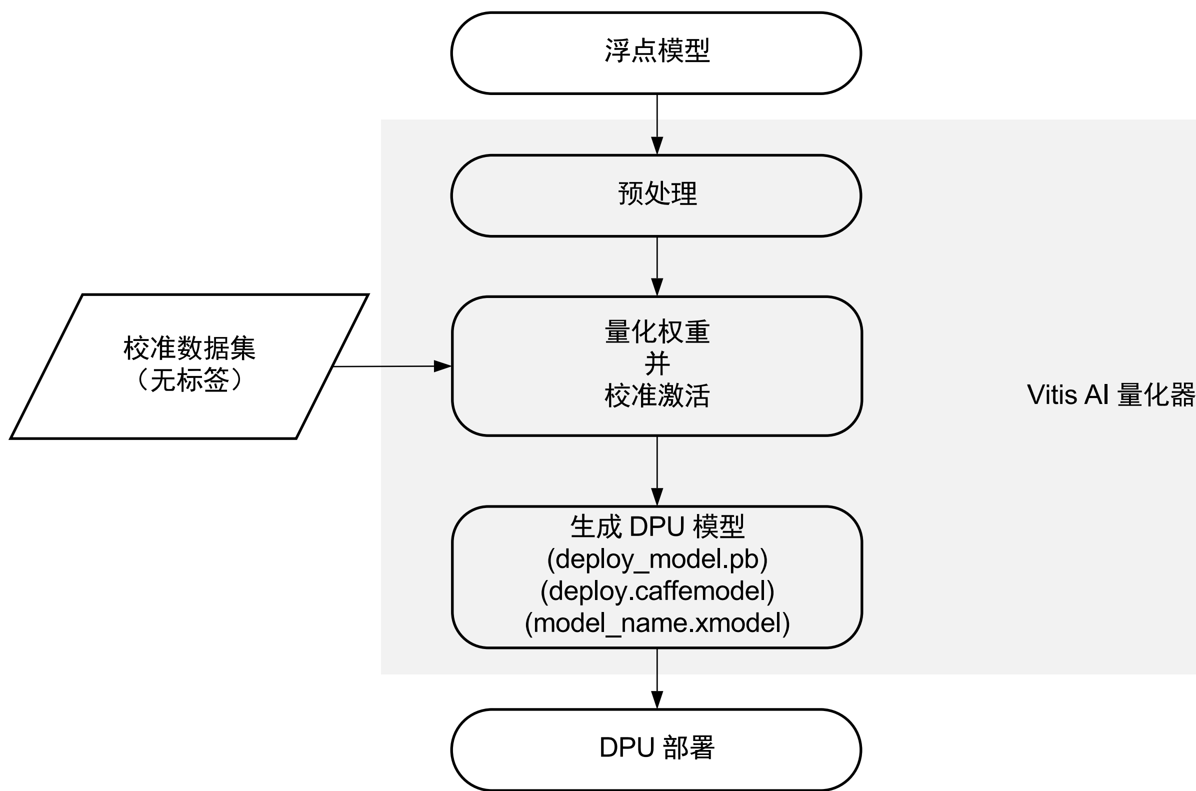
参考资料

1. Markus Nagel 等，《Data-Free Quantization through Weight Equalization and Bias Correction》，arXiv:1906.04721, 2019。
2. Itay Hubara 等，《Improving Post Training Neural Quantization: Layer-wise Calibration and Integer Programming》，arXiv:2006.10518, 2020。

# Vitis AI 量化器流程

下图描述了整个模型量化流程。

图 17：**VAI 量化器工作流程**




**注释：**从 Vitis AI 2.5 起，已弃用 Caffe。如需了解有关 Caffe 的信息，请参阅《Vitis AI 2.0 用户指南》。

Vitis AI 量化器接受浮点模型作为输入，并执行预处理（折叠 batch-norm 并移除推断不需要的节点）。然后，它将权重/偏差和激活量化至给定的位宽。

量化模型前，使用称为“检查器”的步骤对模型进行检查。检查器会输出分区信息，以指示可在相应器件 (DPU/CPU) 上运行的各项运算。总体上，DPU 比 CPU 更快，意图是在 DPU 器件上尽可能运行更多运算符。分区结果包含各项消息用于说明特定运算符无法在 DPU 上执行的原因。这有助于您更好地了解 DPU 的能力，并帮助您调整 DPU 模型。

Vitis AI 量化器需要多次迭代推断来校准激活，以提高量化模型的精度并捕获激活统计数据。这需要一个校准图像数据集作为输入。通常由于不需要反向传播，量化器能够妥善处理 100 到 1000 张校准图像，而无标记的数据集也能发挥作用。

校准后，量化模型变换为 DPU 可部署格式（对于 vai_q_tensorflow，它被称为 deploy_model.pb，对于 vai_q_pytorch，则称为 model_name.xmodel），此格式与 DPU 的数据格式保持一致。然后，Vitis AI 编译器即可编译该模型并将其部署到 DPU。但标准版 TensorFlow 或 PyTorch 框架无法直接接受量化模型。

# TensorFlow 1.x 版本 (vai_q_tensorflow)

## 安装 vai_q_tensorflow

vai_q_tensorflow 有两种安装方法：

#### 使用 Docker 容器安装

Vitis AI 为量化工具（包括 vai_q_tensorflow）提供了 Docker 容器。运行容器后，请激活 Conda 环境 Vitis AI-tensorflow：

```
[docker] $ conda activate vitis-ai-tensorflow
```

如果 Docker 容器内有 package-tensorflow 补丁包，则请在 Docker 容器内安装 Vitis AI-tensorflow 补丁包：

```
[docker] $ sudo env CONDA_PREFIX=/opt/vitis_ai/conda/envs/vitis-ai-
tensorflow/ PATH=/opt/vitis_ai/conda/bin:$PATH conda install
patch_package.tar.bz2
```

#### 使用源代码安装

vai_q_tensorflow 是 AMD 维护的插件工具，适用于 TensorFlow 1.15。它在 Vitis_AI_Quantizer 中开源。要构建 vai_q_tensorflow，请运行以下命令：

```
[host] $ sh build.sh
```

# 运行 vai_q_tensorflow

## 准备浮点模型和相关输入文件

执行 vai_q_tensorflow 前，请确保您的浮点格式的冻结推断 TensorFlow 模型和校准集都已准备就绪，其中应包括下表中列出的文件。

表 6：**vai_q_tensorflow 的输入文件**

| 编号 | 名称 | 描述 |
|---|---|---|
| 1 | frozen_graph.pb | 浮点冻结推断计算图。确保此计算图为推断计算图，而非训练计算图。 |
| 2 | 校准数据集 | 训练数据集的子集，包含 100 到 1000 张图像。 |
| 3 | input_fn | 这是输入函数，用于在训练后量化期间将校准数据集转换为 frozen_graph 的输入数据。通常用于执行数据预处理和数据增广。 |

## 生成冻结推断计算图

使用 TensorFlow 1.x 来训练模型时，该进程会创建一个文件夹，其中包含一个 GraphDef 文件（通常扩展名为 .pb 或 .pbtxt）和一组检查点文件。为便于移动或嵌入式部署，您需要满足以下条件的单个 GraphDef 文件：此文件已冻结，或者其变量已转换为内联常量，使该文件可包含所有必要信息。TensorFlow 提供的 `freeze_graph.py` 可用于处理转换，它随 vai_q_tensorflow 量化器一起自动安装。

命令行用法示例如下：

```
[docker] $ freeze_graph \
    --input_graph  /tmp/inception_v1_inf_graph.pb \
    --input_checkpoint  /tmp/checkpoints/model.ckpt-1000 \
    --input_binary  true \
    --output_graph  /tmp/frozen_graph.pb \
    --output_node_names  InceptionV1/Predictions/Reshape_1
`
```

-input_graph 应为推断计算图，而不是训练计算图。由于推断和部署无需数据预处理和损失函数运算，因此 frozen_graph.pb 应仅包含模型的必要组件。尤其是，Input_fn 应包含数据预处理运算，以便生成正确的输入数据用于训练后量化。

**注释：**此类运算（如，dropout 和 batch norm）在训练阶段与推断阶段中的行为表现不同。对计算图进行冻结时，请确保这些操作在推断阶段执行。例如，使用 tf.layers.dropout/tf.layers.batch_normalization 时可设置 is_training=false 标志。对于使用 tf.keras 的模型，请先调用 tf.keras.backend.set_learning_phase(0)，然后再构建计算图。

**提示：**输入 freeze_graph --help 可获取更多选项。

输入节点和输出节点名称因模型而异，但您可使用 vai_q_tensorflow 量化器来检查和估算这些节点。请参阅以下代码片段示例：

```
[docker] $ vai_q_tensorflow inspect --input_frozen_graph=/tmp/
inception_v1_inf_graph.pb
```

如果计算图包含计算图内预处理和后处理，那么估算的输入和输出节点不得用于量化。这是因为某些操作无法量化，使用 Vitis AI 编译器来编译模型并将其部署到 DPU 时，这类操作可能会导致错误。

另一种获取计算图的输入和输出名称的方法是将计算图可视化。TensorBoard 和 Netron 均可执行此操作。请参阅以下示例，其中使用的是 Netron：

```
[docker] $ pip install netron
[docker] $ netron /tmp/inception_v3_inf_graph.pb
```

## 准备校准数据集和输入函数

校准集通常是训练的子集、确认数据集的子集或者是实际的应用图像（至少包含 100 张图像以保证最优性能）。输入函数是 Python 可导入的函数，用于处理数据预处理。该函数会加载校准数据集，并执行必要的数据预处理步骤。vai_q_tensorflow 量化器可接受 input_fn 用于预处理，但在计算图中不保存 input_fn。但如果预处理子计算图保存到冻结计算图，那么 input_fn 只需从数据集读取图像并返回 feed_dict 即可。

该输入函数遵循 module_name.input_fn_name 格式（例如，my_input_fn.calib_input）。它接受一个表示校准步骤编号的 int 对象，并返回一个 dict 对象，其中包含对应每次调用的 placeholder_name, numpy.Array。在推断期间，会将该对象馈送到模型的占位符节点中。placeholder_name 始终与充当输入数据接收节点的冻结计算图的输入节点相对应。

**注释：**placeholder_name 应替换为接收输入图像的输入节点的实际名称。例如，如果输入占位符节点名为 the_input_node，那么 placeholder_name 应替换为 the_input_node。

vai_q_tensorflow 选项中的 input_nodes 指示冻结计算图中量化开始的位置。placeholder_names 和 input_nodes 选项有时候不同。当冻结计算图包含计算图内预处理时，placeholder_name 表示计算图的输入。但建议将 input_nodes 设为预处理步骤的最后一个节点。请确保 numpy.the array 的形状与对应占位符一致。以下提供了伪代码示例以供参考：

```
$ "my_input_fn.py"
def calib_input(iter):
"""
A function that provides input data for the calibration
  Args:
    iter: A `int` object, indicating the calibration step number
  Returns:
    dict( placeholder_name, numpy.array): a `dict` object, which will be
```

```
fed into the model
"""
  image = load_image(iter)
  preprocessed_image = do_preprocess(image)
  return {"placeholder_name": preprocessed_images}
```

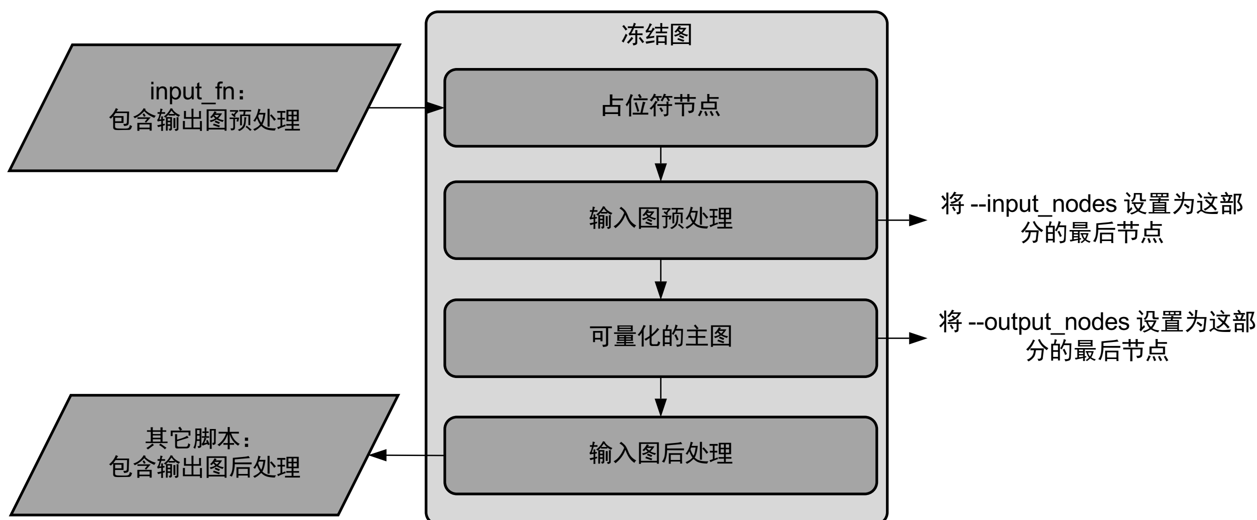
## 使用 vai_q_tensorflow 量化模型

运行以下命令以量化模型：

```
$vai_q_tensorflow quantize \
                --input_frozen_graph  frozen_graph.pb \
                --input_nodes     ${input_nodes} \
                --input_shapes    ${input_shapes} \
                --output_nodes    ${output_nodes} \
                --input_fn  input_fn \
                [options]
```

`input_nodes` 和 `output_nodes` 实参均为量化计算图的输入节点的名称列表。这两个实参分别作为量化的起点和终点。两者间的主计算图将进行量化（如可量化），如下图所示。

图 18：**TensorFlow 量化流程**




建议将 `-input_nodes` 设置为预处理部分的最后一个节点，将 `-output_nodes` 设置为主计算图的最后一个节点，因为预处理和后处理部分的某些操作不可量化。由 Vitis AI 编译器对该模型进行编译并将其部署到 DPU 时，这可能导致错误。

输入节点可能与计算图的占位符节点不同。如果冻结计算图不包含计算图内预处理，则应将占位符节点设置为输入节点。

`input_fn` 应与占位符节点保持一致。

[options] 表示可选参数。最常用的选项如下：

- weight_bit：已量化的权重和偏差的宽度，默认值为 8。
- activation_bit：量化激活的位宽，默认值为 8。

- method：量化方法，0 对应非溢出方法，1 对应最小差值方法，2 对应归一化的最小差值方法。此非溢出方法用于确保量化期间没有任何值达到饱和。结果可能受到离群值的影响。最小差值方法允许饱和以使量化尽可能降低量化差值。对于离群值而言，此方法更稳健，通常生成的范围比非溢出方法更小。

## 生成量化模型

成功执行 vai_q_tensorflow 命令后，会在 ${output_dir} 中生成一个输出文件。quantize_eval_model.pb 用于对 CPU 和 GPU 上的量化模型精度执行评估，并在硬件上对结果进行仿真。

表 7：**vai_q_tensorflow 输出文件**

| 编号 | 名称 | 描述 |
|---|---|---|
| 1 | deploy_model.pb | Vitis AI 编译器的量化模型（TensorFlow 扩展格式），用于执行目标 DPUCZDX8G 实现。 |
| 2 | quantize_eval_model.pb | 要评估的量化模型（同时也是诸如 DPUCAHX8H 和 DPUCADF8H 等大部分 DPU 架构的 Vitis AI 编译器输入）。 |

## （可选）将量化模型导出到 ONNX

量化模型默认采用 TensorFlow 协议缓冲器 (Protobuf) 格式。如需 ONNX 格式的模型，请将 output_format 实参添加到 vai_q_tensorflow 命令：

```
[docker] $ vai_q_tensorflow quantize \
--input_frozen_graph frozen_graph.pb \
--input_nodes ${input_nodes} \
--input_shapes ${input_shapes} \
--output_nodes ${output_nodes} \
--input_fn input_fn \
--output_format onnx \
[options]
```

- output_format：指示量化模型的保存格式，pb 表示保存 TensorFlow 冻结 Protobuf，onnx 表示保存 ONNX 模型。默认值为 pb。

## （可选）评估量化模型

如有脚本可用于评估浮点模型（如 Vitis AI Model Zoo 中的模型），请应用下列 2 项更改以评估量化模型：

- 为浮点评估脚本追加 import vai_q_tensorflow。
- 将脚本中的浮点模型路径替换为量化输出模型：quantize_results/quantize_eval_model.pb。
- 运行修改后的脚本以评估量化模型。

## （可选）转储仿真结果

vai_q_tensorflow 使用量化器生成的 quantize_eval_model.pb 转储仿真结果。这样，您就可以将 CPU/GPU 上的仿真结果与 DPU 上的输出值进行比较。

要转储量化仿真结果，请运行以下命令：

```
[docker] $ vai_q_tensorflow dump \
    --input_frozen_graph  quantize_results/quantize_eval_model.pb \
    --input_fn  dump_input_fn \
    --max_dump_batches 1 \
    --dump_float 0 \
    --output_dir quantize_results
```

用于转储的 input_fn 与用于量化校准的 input_fn 类似，但批次大小通常设置为 1 以便与 DPU 结果保持一致。

成功执行此命令后，会在 ${output_dir} 中生成转储结果。${output_dir} 包含多个文件夹，每个文件夹都包含某一批次的输入数据的转储结果。对于每个量化节点，此结果分别以 *_int8.bin 和 *_int8.txt 格式保存。如果 dump_float 设为 1，则会转储未经量化的节点的结果。 “/” 符号将被替换为 “_” 以便于处理。下表显示了转储结果的部分示例。

表 8：**转储结果示例**

| 批次编号 | 是否量化 | 节点名称 | 已保存的文件 |
|---|---|---|---|
| 1 | 是 | resnet_v1_50/conv1/biases/wquant | {output_dir}/dump_results_1/resnet_v1_50_conv1_biases_wquant_int8.bin<br>{output_dir}/dump_results_1/resnet_v1_50_conv1_biases_wquant_int8.txt |
| 2 | 否 | resnet_v1_50/conv1/biases | {output_dir}/dump_results_2/resnet_v1_50_conv1_biases.bin<br>{output_dir}/dump_results_2/resnet_v1_50_conv1_biases.txt |

## vai_q_tensorflow 量化感知训练

量化感知训练 (QAT) 与浮点模型训练/微调类似。但在 QAT 中，vai_q_tensorflow API 会在训练开始前将浮点计算图转换为量化计算图。典型工作流程如下：

1. 准备：开始 QAT 前，准备下列文件：

   表 9：**vai_q_tensorflow QAT 的输入文件**

   | 编号 | 名称 | 描述 |
   |---|---|---|
   | 1 | 检查点文件 | 作为起点的浮点检查点文件。如果要从头开始训练模型，请忽略此文件。 |
   | 2 | 数据集 | 含标记的训练数据集。 |
   | 3 | 训练脚本 | 用于运行模型的浮点训练/微调的 Python 脚本。 |

2. 评估浮点模型（可选）：在执行量化微调之前，评估浮点检查点文件以检查脚本和数据集的准确性。浮点检查点的精度和损失值也可作为 QAT 的基线。

3. 修改训练脚本：要创建量化训练计算图，请在构建浮点计算图后，修改训练脚本以调用函数。下面给出了 1 个示例：

```
# train.py

# ...

# Create the float training graph
model = model_fn(is_training=True)

# *Set the quantize configurations
import vai_q_tensorflow
q_config = vai_q_tensorflow.QuantizeConfig(input_nodes=['net_in'],
                output_nodes=['net_out'],
                input_shapes=[[-1, 224, 224, 3]])
# *Call Vai_q_tensorflow API to create the quantize training graph

vai_q_tensorflow.CreateQuantizeTrainingGraph(config=q_config)

# Create the optimizer
```

```
optimizer = tf.train.GradientDescentOptimizer()

# start the training/finetuning; you can use sess.run(), tf.train,
tf.estimator, tf.slim and so on
# ...
```

**注释：**您可使用 `import vai_q_tensorflow as decent_q` 来保证与 vai_q_tensorflow 的更低版本的代码（即，`import tensorflow.contrib.decent_q`）兼容

`QuantizeConfig` 包含量化的配置。

部分基本配置（如，`input_nodes`、`output_nodes` 和 `and input_shapes`）必须根据模型结构加以设置。

其他配置（如 `weight_bit`、`activation_bit` 和 `method`）则包含默认值，可按需修改。请参阅 vai_q_tensorflow 用法 以获取所有配置的详细信息。

- `input_nodes/output_nodes`：这两项结合使用，即可判定要量化的子计算图范围。预处理和后处理组件通常不可量化，应置于此范围之外。input_nodes 与 output_nodes 应相同，以使浮点训练计算图与评估计算图之间的量化运算相匹配。

  **注释：**当前不支持含多个输出张量（例如，FIFO）的运算。可添加 tf.identity 节点为 input_tensor 生成别名，以构成含单一输出的输入节点。

- `input_shapes`：对于每个节点，input_nodes 的形状列表必须为 4 维。该信息以逗号分隔，例如，[[1,224,224,3] [1, 128, 128, 1]]；支持 batch_size 为未知大小，例如，[[-1,224,224,3]]。

4. 评估并生成量化模型：在 QAT 之后，使用检查点文件评估量化计算图，并生成冻结模型。方法是在构建浮点评估计算图之后调用以下函数。冻结进程取决于量化评估计算图，因此通常两者一起调用。

   **注释：**`vai_q_tensorflow.CreateQuantizeTrainingGraph` 函数和 `vai_q_tensorflow.CreateQuantizeEvaluationGraph` 函数用于修改 Tensorflow 中的默认计算图。这两个函数必须在不同计算图阶段内分别调用。`vai_q_tensorflow.CreateQuantizeTrainingGraph` 必须在浮点训练计算图上调用，而 `vai_q_tensorflow.CreateQuantizeEvaluationGraph` 则需在浮点评估计算图上调用。调用 `vai_q_tensorflow.CreateQuantizeTrainingGraph` 函数后就无法再调用 `vai_q_tensorflow.CreateQuantizeEvaluationGraph`，因为默认计算图已转换为量化训练计算图。正确的方法是在调用浮点模型创建函数后立即调用该函数。

```
# eval.py

# ...

# Create the float evaluation graph
model = model_fn(is_training=False)

# *Set the quantize configurations
import vai_q_tensorflow
q_config = vai_q_tensorflow.QuantizeConfig(input_nodes=['net_in'],
                output_nodes=['net_out'],
                input_shapes=[[-1, 224, 224, 3]])
# *Call Vai_q_tensorflow API to create the quantize evaluation graph

vai_q_tensorflow.CreateQuantizeEvaluationGraph(config=q_config)
# *Call Vai_q_tensorflow API to freeze the model and generate the deploy
model

vai_q_tensorflow.CreateQuantizeDeployGraph(checkpoint="path to
checkpoint folder", config=q_config)

# start the evaluation; You can use sess.run, tf.train, tf.estimator,
tf.slim and so on
# ...
```

## 生成的文件

执行完上述步骤后，将在 `${output_dir}` 生成以下文件。

表 10：生成的文件信息

| 名称 | TensorFlow 是否兼容 | 用法 | 描述 |
|---|---|---|---|
| quantize_train_graph.pb | 是 | 训练 | 量化训练计算图。 |
| quantize_eval_graph_{suffix}.pb | 是 | 使用检查点进行评估 | 量化评估计算图，内部冻结有量化信息。该文件有权重，应与检查点文件一起进行评估。 |
| quantize_eval_model_{suffix}.pb | 是 | 1：评估<br>2：转储<br>3：输入 VAI 编译器 (DPUCAHX8H) | 冻结的量化评估计算图，其中冻结有检查点中的权重和量化信息。它可用于在主机上评估量化模型，或者对每一层的输出进行转储以供与 DPU 输出进行交叉比对检查。XIR 编译器使用它作为输入。 |

后缀包含来自检查点文件的迭代信息和日期信息。例如，如果检查点文件为“model.ckpt-2000.*”，而日期为 20200611，那么后缀为“2000_20200611000000”。

## 适用于 TensorFlow 1.x 的 QAT API

在 Python 包 `vai_q_tensorflow` 中有 3 个 API 可用于 QAT。

### vai_q_tensorflow.CreateQuantizeTrainingGraph (config)

通过在默认计算图上直接重写，将浮点训练计算图转换为量化训练计算图。

#### 实参

- `config`：`vai_q_tensorflow.QuantizeConfig` 对象，其中包含量化配置。

### vai_q_tensorflow.CreateQuantizeEvaluationGraph(config)

将浮点评估计算图转换为量化评估计算图。这是通过对默认计算图进行就地重写来完成的。

#### 实参

- `config`：`vai_q_tensorflow.QuantizeConfig` 对象，其中包含量化配置。

### vai_q_tensorflow.CreateQuantizeDeployGraph(checkpoint, config)

将检查点冻结到量化评估计算图中。

#### 实参

- `checkpoint`：`string` 对象，指定检查点文件夹或文件的路径。
- `config`：`vai_q_tensorflow.QuantizeConfig` 对象，包含量化所需的配置。

## QAT 技巧

以下提供了一些 QAT 技巧。

- Keras 模型：

  对于 Keras 模型，创建浮点训练计算图前，请设置 `backend.set_learning_phase(1)`，创建浮点评估计算图前，请设置 `backend.set_learning_phase(0)`。此外，应先调用 `backend.clear_session()`，而后再调用 `backend.set_learning_phase()`。Tensorflow1.x QAT API 是专为 TensorFlow 原生训练 API 而设计的。在 QAT 内使用 Keras `model.fit()` API 可能导致出现某些“nodes not executed”（节点未执行）问题。建议在 Tensorflow2 量化工具内将 QAT API 与 Keras API 搭配使用。

- 丢弃：实验证明 QAT 在没有丢弃 (dropout) 运算的情况下更有效。该工具当前不支持执行含 dropout 的微调，在运行 QAT 前，应移除或禁用 dropout。方法是使用 tf.layers 时设置 `is_training=false`，或者使用 tf.keras.layers 时调用 `tf.keras.backend.set_learning_phase(0)`。

- 超参数：QAT 类似于浮点模型训练/微调，因此 QAT 中同样需要使用浮点模型训练/微调中使用的技术。需要微调的重要参数包括优化器类型和学习率曲线等。

## 转换为 Float16 或 BFloat16

vai_q_tensorflow 支持为浮点模型进行数据类型转换，受支持的模型包括 Float16、BFloat16、Float 和 Double。您可将 convert_datatype 实参添加到 vai_q_tensorflow 命令以完成此转换。

```
$vai_q_tensorflow quantize \
--input_frozen_graph frozen_graph.pb \
--input_nodes ${input_nodes} \
--input_shapes ${input_shapes} \
--output_nodes ${output_nodes} \
--input_fn input_fn \
--convert_datatype 1 \
[options]
```

- convert_datatype：int。用于指定要转换到的目标数据类型。1 对应 Float16、2 对应 Double、3 对应 BFloat16，4 对应 Float。默认值为 0。

**注释：**BatchNorm 操作会提前折叠以实现转换。

## vai_q_tensorflow 支持的运算和 API

下表列出受支持的 vai_q_tensorflow 运算和 API。

表 11：**vai_q_tensorflow 支持的运算和 API**

| 类型 | 运算类型 | tf.nn | tf.layers | tf.keras.layers |
|---|---|---|---|---|
| Convolution | Conv2D<br>DepthwiseConv2dNative | atrous_conv2d<br>conv2d<br>conv2d_transpose<br>depthwise_conv2d_native<br>separable_conv2d | Conv2D<br>Conv2DTranspose<br>SeparableConv2D | Conv2D<br>Conv2DTranspose<br>DepthwiseConv2D<br>SeparaleConv2D |
| Fully Connected | MatMul | / | Dense | Dense |
| BiasAdd | BiasAdd<br>Add | bias_add | / | / |
| Pooling | AvgPool<br>Mean<br>MaxPool | avg_pool<br>max_pool | AveragePooling2D<br>MaxPooling2D | AveragePooling2D<br>MaxPool2D |
| Activation | ReLU<br>ReLU6<br>Sigmoid<br>Swish<br>Hard-sigmoid<br>Hard-swish | relu<br>relu6<br>leaky_relu<br>swish | / | ReLU<br>Leaky ReLU |
| BatchNorm[#1] | FusedBatchNorm | batch_normalization<br>batch_norm_with_global_normalization<br>fused_batch_norm | BatchNormalization | BatchNormalization |
| Upsampling | ResizeBilinear<br>ResizeNearestNeighbor | / | / | UpSampling2D |

表 11：**vai_q_tensorflow 支持的运算和 API** (续)

| 类型 | 运算类型 | tf.nn | tf.layers | tf.keras.layers |
|---|---|---|---|---|
| Concat | Concat<br>ConcatV2 | / | / | Concatenate |
| 其他 | Placeholder<br>Const<br>Pad<br>Squeeze<br>Reshape<br>ExpandDims<br>Max<br>Transpose | dropout[#2]<br>softmax[#3]<br>depth_to_space | Dropout[#2]<br>Flatten | Input<br>Flatten<br>Reshape<br>Zeropadding2D<br>Softmax |

**注释：**

1. 仅支持 Conv2D/DepthwiseConv2D/Dense+BN。折叠 BN 可提高推断性能。
2. 删除 dropout 可提高推断性能。
3. vai_q_tensorflow 不会量化 softmax 输出。

**注释：**Vitis AI 量化器支持的运算符列表并不是模型部署的唯一限制因素，您还应查看所选 DPU 架构的运算符支持情况。DPU 不支持的运算符可在 CPU 上执行。如需了解更多信息，请参阅 受支持的运算符和 DPU 限制。

## vai_q_tensorflow 用法

下表显示了 `vai_q_tensorflow` 选项。

表 12：**vai_q_tensorflow 选项**

| 名称 | 类型 | 描述 |
|---|---|---|
| **常用配置** | | |
| --input_frozen_graph | 字符串 | 适用于浮点模型的 TensorFlow 冻结的 GraphDef 推断文件，它用于训练后量化。 |
| --input_nodes | 字符串 | 指定量化计算图的输入节点名称列表，以逗号分隔，搭配 `-output_nodes` 一起使用。输入节点和输出节点即为量化的起点和终点。两者间的子计算图在可量化情况下就会加以量化。<br><br>**建议：**将 `-input_nodes` 设置为预处理的最后节点，并将 `-output_nodes` 设置为后处理的最后节点，因为预处理和后处理所需的某些运算不可量化。由 Vitis AI 编译器对该模型进行编译并将其部署到 DPU 时，这些节点可能导致错误。输入节点可能与计算图的占位符节点不同。 |
| --output_nodes | 字符串 | 指定量化计算图的输出节点名称列表，以逗号分隔，并与 `-input_nodes` 结合使用。输入节点和输出节点即为量化的起点和终点。两者间的子计算图在可量化情况下就会加以量化。<br><br>**建议：**将 `-input_nodes` 设置为预处理的最后节点，并将 `-output_nodes` 设置为后处理的最后节点，因为预处理和后处理所需的某些运算不可量化。由 Vitis AI 编译器对该模型进行编译并将其部署到 DPU 时，这些节点可能导致错误。 |
| --input_shapes | 字符串 | 指定输入节点的形状列表。它必须设为每个节点的四维形状，并以逗号分隔。例如，1,224,224,3。针对 batch_size 支持使用未知大小，例如，224,224,3。对于多个输入节点，请分配每个节点的形状列表（以 , 分隔），例如，? 224,224,3:? 300,300,1. |
| --input_fn | 字符串 | 为搭配校准数据集使用的计算图提供输入数据。该函数遵循 `module_name.input_fn_name` 格式（例如，`my_input_fn.input_fn`）。`-input_fn` 应采用 `int` 对象作为输入以表示校准步骤，并且应返回成对 ("placeholder_node_name, numpy.Array") 形式的 dict 作为每次调用的对象。随后，将该对象馈送至模型的占位符运算。<br><br>例如，您可将 `-input_fn` 分配给 `my_input_fn.calib_input` ，并创建 `calib_input` 函数 `in my_input_fn.py`，如下所示：<br><br>`def calib_input_fn:`<br>`# read the image and do some preprocessing`<br>`    return {"placeholder_1": input_1_nparray,`<br>`"placeholder_2": input_2_nparray}`<br><br>**注释：**您无需在 `input_fn` 中再次执行计算图内预处理，因为量化期间会保留 `-input_nodes` 之前的子计算图。移除预定义的输入函数（包括默认函数和随机函数），因为这些函数不常用。不包含在计算图文件中的预处理部分，应在 `input_fn` 中进行处理。 |
| **量化配置** | | |
| --weight_bit | Int32 | 指定已量化的权重和偏差的位宽。<br>默认值：8 |
| --activation_bit | Int32 | 指定已量化的激活的位宽。<br>默认值：8 |

表 12：**vai_q_tensorflow** 选项 (续)

| 名称 | 类型 | 描述 |
| --- | --- | --- |
| --nodes_bit | 字符串 | 指定节点的位宽。节点名称与位宽构成一对参数（以冒号相连），并且这些参数以逗号分隔。指定 conv op 名称时，只有 `vai_q_tensorflow` 会使用指定的位宽量化 conv op 的权重。例如，conv1/Relu:16,conv1/weights:8,conv1:16。 |
| --method | Int32 | 指定量化方法。<br>・ 0：非溢出方法，量化期间不会有任何值达到饱和。易受离群值影响。<br>・ 1：最小差值方法，支持饱和以使量化尽可能降低量化差值。对于离群值容限更高。通常相比非上溢方法，此方法最终范围更小。<br>・ 2：使用最小差值方法搭配逐通道策略。它支持在量化期间较大值达到饱和，以获得较小的量化误差。对逐通道权重应用特定策略。它比方法 0 慢，但对离群值有更高的耐受力。<br>默认值：1 |
| --nodes_method | 字符串 | 指定节点的方法。节点名称与方法构成一对参数（以冒号相连），并且这些参数对以逗号分隔。指定 conv op 名称时，只有 `vai_q_tensorflow` 会使用指定方法来量化 conv op 的权重，例如，“conv1/Relu:1,depthwise_conv1/weights:2,conv1:1”。 |
| --calib_iter | Int32 | 指定校准迭代。校准图像总数 = calib_iter * batch_size。<br>默认值：100 |
| --ignore_nodes | 字符串 | 指定量化期间要忽略的节点列表。量化期间，已忽略的节点将保留不予量化。 |
| --skip_check | Int32 | 如果设为 1，则跳过浮点模型检查。当仅对输入模型某一部分进行量化时，此项很有用。<br>范围：[0, 1]<br>默认值：0 |
| --align_concat | Int32 | 指定用于对齐 concat 节点的输入量化位置的策略。<br>0：对齐所有 concat 节点<br>1：对齐输出 concat 节点<br>2：禁用对齐<br>默认值：0 |
| --align_pool | Int32 | 指定用于对齐 maxpool/avgpool 节点的输入量化位置的策略。<br>0：对齐所有 maxpool/avgpool 节点<br>1：对齐输出 maxpool/avgpool 节点<br>2：禁用对齐<br>默认值：0 |
| --simulate_dpu | Int32 | 设为 1 即可启用 DPU 仿真。DPU 的部分运算的行为与 TensorFlow 不尽相同。例如，LeakyRelu 和 AvgPooling 中的除法被移位所替代，因此 DPU 输出与 CPU/GPU 输出之间可能存在些许差异。如果此标志设为 1，那么 vai_q_tensorflow 量化器会对这些运算的行为进行仿真。<br>范围：[0, 1]<br>默认值：1 |

表 12：**vai_q_tensorflow** 选项 (续)

| 名称 | 类型 | 描述 |
|---|---|---|
| --adjust_shift_bias | Int32 | 指定 DPU 编译器的移位偏差检查和调整策略。<br>0：禁用移位偏差检查和调整<br>1：在静态约束下启用<br>2：在动态约束下启用<br>默认值：1 |
| --adjust_shift_cut | Int32 | 指定 DPU 编译器的移位切割检查和调整策略。<br>0：禁用移位切割检查和调整<br>1：在静态约束下启用<br>默认值：1 |
| --arch_type | 字符串 | 指定固定神经元的架构类型。DEFAULT 表示权重和激活的量化范围是 [-128, 127]。“DPUCADF8H”表示权重量化范围为 [-128, 127]，而激活的量化范围则为 [-127, 127] |
| --output_dir | 字符串 | 指定量化结果的保存目录。<br>默认值：“./quantize_results” |
| --max_dump_batches | Int32 | 指定用于转储的最大批次数。<br>默认值：1 |
| --dump_float | Int32 | 如果设为 1，那么会转储浮点权重和激活。<br>范围：[0, 1]<br>默认值：0 |
| --dump_input_tensors | 字符串 | 指定当计算图入口并非占位符时，此计算图的输入张量名称。将占位符添加到 `dump_input_tensor`，以便 `input_fn` 可馈送数据。 |
| --scale_all_avgpool | Int32 | 设为 1 即启用 AvgPooling 运算的比例输出以仿真 DPU。仅当 kernel_size <= 64 时才会执行缩放。此运算不影响特殊情况，例如 kernel_size=3,5,6,7,14<br>默认值：1 |
| --do_cle | Int32 | 1：启用跨层均衡的实现，以便调整权重分布<br>0：跳过跨层均衡运算<br>默认值：0 |
| --replace_relu6 | Int32 | 仅限 do_cle=1 时可用<br>1：将 ReLU6 替换为 ReLU<br>0：跳过替换<br>默认值：1 |
| --replace_sigmoid | Int32 | 1：将 sigmoid 替换为 hard-sigmoid<br>0：跳过替换<br>默认值：0 |
| --replace_softmax | Int32 | 1：将 softmax 替换为 hard-softmax<br>0：跳过替换<br>默认值：0 |

表 12：**vai_q_tensorflow 选项 (续)**

| 名称 | 类型 | 描述 |
| --- | --- | --- |
| --convert_datatype | Int32 | 4：执行 BN 折叠，并转换为数据类型 fp32<br>3：执行 BN 折叠，并转换为数据类型 bfloat16<br>2：执行 BN 折叠，并转换为数据类型 double<br>1：执行 BN 折叠，并转换为数据类型 fp16<br>0：跳过转换<br>默认值：0 |
| --output_format | 字符串 | 指示量化模型的保存格式，pb 表示保存 tensorflow 冻结 pb，onnx 表示保存 ONNX 模型。<br>默认值：'pb' |
| **会话配置** | | |
| --gpu | 字符串 | 指定用于量化的 GPU 器件 ID，以逗号分隔。 |
| --gpu_memory_fraction | 浮点值 | 指定用于量化的 GPU 内存小数（介于 0-1 之间）。<br>默认值：0.5 |
| **其他** | | |
| --help | | 显示所有可用的 vai_q_tensorflow 选项。 |
| --version | | 显示 vai_q_tensorflow 版本信息。 |

**示例**

```
show help: vai_q_tensorflow --help
quantize:
vai_q_tensorflow quantize --input_frozen_graph frozen_graph.pb \
                          --input_nodes inputs \
                          --output_nodes predictions \
                          --input_shapes ?,224,224,3 \
                          --input_fn my_input_fn.calib_input
dump quantized model:
vai_q_tensorflow dump --input_frozen_graph quantize_results/
quantize_eval_model.pb \
                      --input_fn my_input_fn.dump_input
```

请参阅 AMD Model Zoo 以获取有关 TensorFlow 模型量化的更多示例。

# vai_q_tensorflow 错误代码

表 13：**vai_q_tensorflow 错误代码**

| 错误代码 | 原因 | 解决方案 |
| --- | --- | --- |
| Quantize_TF1_Invalid_Input | 未找到指定的 input_frozen_graph 文件 | 请验证 `input_frozen_graph` 是否正确以及此文件是否存在。 |
| Quantize_TF1_Invalid_Bitwidth | 指定的 `nodes_bit` 值无效，例如，小于 1 | 请验证 `nodes_bit` 内容是否正确。 |
| Quantize_TF1_Invalid_Method | 指定的 `method` 值无效，不在 [0,1,2] 内的范围内 | 验证 `method` 值是否正确。 |
| Quantize_TF1_Length_Mismatch | 指定的 `input_shapes` 无效，例如，与 input_nodes 不匹配、并非 4 维，或者包含非整数元素 | 验证 `input_shapes` 的形状是否正确并匹配 `input_nodes`。 |

表 13：**vai_q_tensorflow 错误代码** (续)

| 错误代码 | 原因 | 解决方案 |
|---|---|---|
| Quantize_TF1_Invalid_Input_Fn | 指定的 `input_fn` 模块导入失败 | 请验证 `input_fn` 是否正确，并确保正确实现该函数。 |
| Quantize_TF1_Invalid_Target_Dtype | 指定的 `convert_datatype` 值无效，不在 [0,1,2,3,4] 内的范围内 | 验证 `convert_datatype` 值是否正确。 |
| Quantize_TF1_Unsupported_Op | 转换数据类型期间，遇到不受支持的运算符，例如，FusedBatchNorm | 替换不受支持的运算符。 |

# TensorFlow 2.x 版本 (vai_q_tensorflow2)

## 安装 vai_q_tensorflow2

可通过以下 3 种方式来安装 vai_q_tensorflow2：

### 使用 Docker 容器安装

Vitis AI 为量化工具（包括 vai_q_tensorflow）提供了 Docker 容器。运行容器后，请激活 Vitis AI-tensorflow2 conda 环境。

```
[docker] $ conda activate vitis-ai-tensorflow2
```

如有补丁包，请在 Docker 容器内部安装 Vitis AI-tensorflow2 补丁包。

```
# [optional]
[docker] $ sudo env CONDA_PREFIX=/opt/vitis_ai/conda/envs/vitis-ai-
tensorflow2/ PATH=/opt/vitis_ai/conda/bin:$PATH conda install
patch_package.tar.bz2
```

### 使用 Wheel Package 从源代码安装

vai_q_tensorflow2 是 TensorFlow 模型最优化工具集的复刻。它在 Vitis_AI_Quantizer 中开源。要构建 vai_q_tensorflow2，请运行以下命令：

```
[host] $ sh build.sh
[host] $ pip install pkgs/*.whl
```

### 使用 Conda Package 从源代码安装

**重要提示！**这需要 Anaconda。

```
# CPU-only version
[host] $ conda build vai_q_tensorflow2_cpu_feedstock --output-folder ./
conda_pkg/
# GPU version
```

```
[host] $ conda build vai_q_tensorflow2_gpu_feedstock --output-folder ./
conda_pkg/
# Install conda package on your machine
[host] $ conda install --use-local ./conda_pkg/linux-64/*.tar.bz2
```

## 检查浮点模型

`VitisInspector` 作为一个帮助程序工具，用于检查浮点模型、显示给定 DPU 目标架构的分区结果，此外还提供某些层未能映射到 DPU 的原因的指示信息。如无 `target`，您只能显示部分与目标无关的通用检查结果。分配 `target` 即可获取更多详细的检查结果。

**注释：**由于 DPU 限制，仅限默认 `pof2s` 量化策略才能使用此功能特性。

以下代码显示了模型检查方式：

```
model = tf.keras.models.load_model('float_model.h5')
from tensorflow_model_optimization.quantization.keras import vitis_inspect
inspector = vitis_inspect.VitisInspector(target="DPUCADF8H_ISA0")
inspector.inspect_model(model,
                        plot=True,
                        plot_file="model.svg",
                        dump_results=True,
                        dump_results_file="inspect_results.txt",
                        verbose=0)
```

- target：string 或 None。表示此模型部署到的目标 DPU。它可采用名称字符串 (DPUCAHX8L_ISA0)、JSON 文件路径（例如，./U50/arch.json）或指纹。如果设为 None，则不应用目标，仅显示部分与目标无关的通用检查结果。默认值为 None。
- model：tf.keras.Model 实例。表示要检查的浮点模型。浮点模型应包含具体的输入形状。以具体的输入形状来构建该模型，或者以 `input_shape` 实参调用 inspect_model。
- input_shape：list(int)、list(list(int))、tuple(int) 或 dictionary(int)。包含每个输入层的输入形状。如不设置，则使用输入层中的默认形状信息。使用多个输入的形状的列表，例如，inspect_model(model, input_shape=[1, 224, 224, 3]) 或 inspect_model(model, input_shape=\[[None, 224, 224, 3], [None, 64, 1]])。所有维度都应包含具体的值，且 batch_size 维度应为 None 或 int。如果模型的输入形状是类似 [None, None, None, 3] 的变量，请指定类似 [None, 224, 224, 3] 的形状，以便生成最终量化模型。默认值为 None。
- plot：bool。表示是否绘制 `graphviz` 的模型检查结果并将图像保存到磁盘。如需将模型检查结果可视化，并提供某些修改提示，那么此项很有用。

  **注释：**仅部分输出文件类型能显示提示，例如，`.svg`。默认值为 False。
- plot_file：字符串。表示绘制模型时的模型图像文件的文件路径。默认值为 `model.svg`。
- dump_results：bool。表示是否转储检查结果并将文本保存到磁盘。相比于屏幕日志记录，它能将更详细的逐层结果转储到文本文件。默认值为 False。
- dump_results_file：string。表示检查结果文本文件的文件路径。默认值为 `inspect_results.txt`。
- verbose：int。表示日志记录的详细程度。verbose 值越高，显示的日志记录结果越详细。默认值为 0。

# 运行 vai_q_tensorflow2

TensorFlow2 量化器支持使用两种方法来量化深度学习模型：

- 训练后量化 (PTQ)：PTQ 是一种将预训练的浮点模型转换为量化模型的技术，模型精度损失极小。要执行 PTQ，就需要一个代表性数据集来对浮点模型运行多批次的推断，这有助获取激活的分布。此进程也称为量化校准。
- 量化感知训练 (QAT)：QAT 用于在模型量化过程中对前向传递和后向传递中的量化误差进行建模。使用 QAT 时，建议从已显示出良好精度的浮点预训练模型开始，而不要从头开始。

## 准备浮点模型和校准集

运行 vai_q_tensorflow2 前，请确保浮点模型和校准集已准备就绪，其中包括下表中列出的文件。

表 14：**vai_q_tensorflow2 的输入文件**

| 编号 | 名称 | 描述 |
|---|---|---|
| 1 | 浮点模型 | TensorFlow 2 浮点模型，采用 h5 格式或已保存的模型格式。 |
| 2 | 校准数据集 | 一小部分训练数据集或确认数据集，用于表示输入数据分发。通常，100 到 1000 张图像足以满足校准需求。 |

## 使用 vai_q_tensorflow2 API 执行量化

以下代码展示了如何利用 vai_q_tensorflow2 API 执行训练后量化。您可在此处找到完整示例。

```
model = tf.keras.models.load_model('float_model.h5')
from tensorflow_model_optimization.quantization.keras import vitis_quantize
quantizer = vitis_quantize.VitisQuantizer(model)
quantized_model = quantizer.quantize_model(calib_dataset=calib_dataset,
                                           calib_steps=100,
                                           calib_batch_size=10,
                                           **kwargs)
```

- calib_dataset：

  `calib_dataset` 是校准进程中使用的代表性校准数据集。它可以全部或部分衍生自 `eval_dataset`、`train_dataset` 或其他数据集。

- calib_steps：`calib_steps` 表示校准步骤总数。默认设为 None。如果 `calib_dataset` 为 `tf.data dataset`、生成器或 `keras.utils.Sequence` 实例且 steps 为 None，校准会持续直至数据集完全耗尽为止。阵列输入不支持此实参。
- calib_batch_size：`calib_batch_size` 用于判定校准期间使用的每批次样本数。如果 `calib_dataset` 为数据集、生成器或 `keras.utils.Sequence` 实例形式，则批次大小由数据集本身控制。但如果 `calib_dataset` 为 `numpy.array` 对象形式，则默认批次大小为 32。
- input_shape：list(int)、list(list(int))、tuple(int) 或 dictionary(int)。包含每个输入层的输入形状。如不显式设置，则使用输入层中的默认形状信息。使用多个输入的形状的列表，例如，input_shape=[1, 224, 224, 3] 或 input_shape=[[None, 224, 224, 3], [None, 64, 1]]。所有维度都应包含具体的值，且 batch_size 维度应为 None 或 int。如果模型的输入形状是类似 [None, None, None, 3] 的变量，那么您需指定类似 [None, 224, 224, 3] 的形状，以便生成最终量化模型。

- **kwargs：`**kwargs` 表示用户定义的量化策略配置字典。它支持您改写默认内置量化策略。例如，设置 `bias_bit=16` 会量化含 16 位量化器的所有偏差。如需了解有关用户定义的配置的更多信息，请参阅 vai_q_tensorflow2 用法 部分。

## （可选）vai_q_tensorflow2 快速微调

通常量化后存在少许精度损失，但对于特定网络（如 MobileNet），精度损失可能较为显著。为此，快速微调使用 AdaQuant 算法，利用未标记的校准数据集逐层调整权重和量化参数，以提升特定模型的精度。

虽然快速微调所需的时间比普通 PTQ 更长（由于 calib_dataset 较小，所用时间仍比 QAT 短得多），但默认处于关闭状态。不过，如果遇到精度问题，可将其启用来提高性能。推荐工作流程是尝试 PTQ 而不进行快速微调，如果对精度不满意，则可尝试量化。

QAT 是改善精度的另一种方法，但它需要更多时间并且依赖于训练数据集。要在训练后量化期间激活快速微调，请设置 `include_fast_ft=True`。

```
quantized_model = quantizer.quantize_model(calib_dataset=calib_dataset,
calib_steps=None, calib_batch_size=None, include_fast_ft=True,
fast_ft_epochs=10)
```

这里，

- `include_fast_ft` 用于判定是否执行快速微调。
- `fast_ft_epochs` 表示每层的微调轮数。

## 保存量化模型

量化模型对象属于标准 `tf.keras` 模型对象。可通过运行以下命令来保存该对象：

```
quantized_model.save('quantized_model.h5')
```

生成的 `quantized_model.h5` 文件可馈送给 vai_c_tensorflow 编译器，然后部署到 DPU 上。

## （可选）将量化模型导出到 ONNX

以下代码展示了如何利用 vai_q_tensorflow2 API 执行训练后量化并将量化模型导出到 ONNX：

```
model = tf.keras.models.load_model('float_model.h5')
from tensorflow_model_optimization.quantization.keras import vitis_quantize
quantizer = vitis_quantize.VitisQuantizer(model)
quantized_model = quantizer.quantize_model(calib_dataset=calib_dataset,
                                           output_format='onnx',
                                           onnx_opset_version=11,
                                           output_dir='./quantize_results',
                                           **kwargs)
```

- output_format：string。指示量化模型的保存格式。选项包括：
  - '' 表示跳过保存
  - h5 表示保存 .h5 文件
  - tf 表示保存 saved_model 文件
  - onnx 表示保存 ONNX 文件

默认值为 ''。

- onnx_opset_version：int。表示 ONNX opset 版本。仅当 output_format 设为 'onnx' 时才生效。默认值为 11。
- output_dir：string。指示量化模型的保存目录。默认值为 "./quantize_results"。

## （可选）评估量化模型

如有脚本可用于评估浮点模型（如 AMD Model Zoo 中的模型），则可将浮点模型文件替换为量化模型用于评估。您必须导入 Vitis_quantize 模块以确保支持自定义的量化层。例如：

```
from tensorflow_model_optimization.quantization.keras import vitis_quantize
quantized_model = tf.keras.models.load_model('quantized_model.h5')
```

随后，只需评估量化模型即可，像浮点模型一样。例如：

```
quantized_model.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy()
,
    metrics= keras.metrics.SparseTopKCategoricalAccuracy())
quantized_model.evaluate(eval_dataset)
```

## （可选）转储仿真结果

有时在部署量化模型后，有必要将 CPU/GPU 上的仿真结果与 DPU 上的输出值进行比对。您可通过 vai_q_tensorflow2 的 `VitisQuantizer.dump_model` API 使用量化模型来转储仿真结果。

```
from tensorflow_model_optimization.quantization.keras import vitis_quantize
quantized_model = keras.models.load_model('./quantized_model.h5')
vitis_quantize.VitisQuantizer.dump_model(model=quantized_model,
                                         dataset=dump_dataset,
                                         output_dir='./dump_results')
```

**注释：**确保 dump_dataset 的 `batch_size` 所设 batch_size 与目标器件上用于 DPU 调试的值相同。对于 DPU 调试，建议使用 CPU 仿真结果，因为 GPU 结果可能会表现出非确定性行为，并在浮点值计算方面存在细微差别。

执行此命令后，会在 `${dump_output_dir}` 中生成转储结果。每个量化层的权重和激活结果都会分别以 *.bin 和 *.txt 格式来加以保存。如果该层的输出未经量化（例如，softmax 层），那么浮点激活结果将保存在 `*_float.bin` 和 `*_float.txt` 文件中。"/"符号将被替换为"_"以便于处理。下表显示了转储结果的部分示例。

**表 15：转储结果示例**

| 批次编号 | 是否量化 | 层名 | 已保存的文件 | | |
|---|---|---|---|---|---|
| | | | 权重 | 偏差 | Activation |
| 1 | 是 | resnet_v1_50/conv1 | {output_dir}/dump_results_weights/quant_resnet_v1_50_conv1_kernel.bin<br>{output_dir}/dump_results_weights/quant_resnet_v1_50_conv1_kernel.txt | {output_dir}/dump_results_weights/quant_resnet_v1_50_conv1_bias.bin<br>{output_dir}/dump_results_weights/quant_resnet_v1_50_conv1_bias.txt | {output_dir}/dump_results_0/quant_resnet_v1_50_conv1.bin<br>{output_dir}/dump_results_0/quant_resnet_v1_50_conv1.txt |

表 15：**转储结果示例** (续)

| 批次编号 | 是否量化 | 层名 | 已保存的文件 | | |
|---|---|---|---|---|---|
| | | | 权重 | 偏差 | Activation |
| 2 | 否 | resnet_v1_50/softmax | 不适用 | 不适用 | {output_dir}/dump_results_0/quant_resnet_v1_50_softmax_float.bin<br>{output_dir}/dump_results_0/quant_resnet_v1_50_softmax_float.txt |

**注释：**DPU 实现中的所有输入和激活的舍入模式均为“HALF_UP”。在实现中使用其他舍入模式可能导致轻微位级不匹配，并附带转储结果。

## 量化策略配置

本节提供了一些默认量化策略，但您必须自定义量化配置以满足不同目标或实现更好的性能。例如，特定的目标设备可能要求将偏差量化为 32 位，而其他设备则可能要求对特定的模型部件进行量化。本节演示了如何配置量化器来满足此类要求。

#### 量化策略

量化工具支持您从三个主要方面进行配置：量化工具流水线、模型的哪些部分需要量化以及如何执行量化。在 `quantize_strategy` 中对这些方面进行定义，此 JSON 文件包含以下配置：

- pipeline_config：

  这些配置用于控制量化工具的工作流水线，包括量化期间的一些优化。此类优化例子包括决定是否折叠 Conv2D + BatchNorm 层，以及是否采用 `Cross-Layer-Equalization` 算法和其他优化。它也可进一步拆分为 `optimize_pipeline_config`、`quantize_pipeline_config`、`refine_pipeline_config` 和 `finalize_pipeline_config`。
- quantize_registry_config：这些配置用于控制有哪些层类型可量化、量化 op 的插入位置以及要插入的量化 op 的种类。它包括某些层专用的配置和用户定义的全局配置。

以下是 Conv2D 层的配置示例：

```
{
  "layer_type": "tensorflow.keras.layers.Conv2D",
  "quantizable_weights": ["kernel"],
  "weight_quantizers": [
  {
    "quantizer_type": "Pof2SQuantizer",
    "quantizer_params": {"bit_width": 8,"method":0, "round_mode": 1,
"symmetry": true, "per_channel": true, "channel_axis": -1, "narrow_range":
False}
  ],
  "quantizable_biases": ["bias"],
  "bias_quantizers": [
  {
    "quantizer_type": "Pof2SQuantizer",
    "quantizer_params": {"bit_width": 8,"method":0, "round_mode": 1,
"symmetry": true, "per_channel": false, "channel_axis": -1, "narrow_range":
False}
  ],
```

```
  "quantizable_activations": ["activation"],
  "activation_quantizers": [
  {
    "quantizer_type": "FSQuantizer",
    "quantizer_params": {"bit_width": 8, "method":2,
"method_percentile":99.9999, "round_mode": 1, "symmetry": true,
"per_channel": false, "channel_axis": -1}
  ]
}
```

您可使用此量化配置对 Conv2D 层的权重、偏置和激活进行量化。权重和偏差使用的是 Pof2SQuantizer（2 的幂值比例量化器），激活使用的则是 FSQuantizer（浮点比例量化器）。您可在单一层内为不同对象采用不同量化器。

**注释：**`Quantizer` 指的是应用于这些配置中的每个对象的量化运算。它取浮点张量作为输入，并生成经量化的张量作为输出。量化进程取“fake”一词，表示将输入量化为整数，然后反量化为浮点数。

### 使用内置量化策略

您可使用 dump_quantize_strategy 获取当前量化策略的 JSON 文件。为简便说明，我们提供了 4 种类型的内置量化策略，这些策略适用于各种常见用例，您可按需对其进行扩展或改写以满足特定要求。这些内置量化策略包括：

- `pof2s`：2 的幂值比例量化，当前主要用于 DPU 目标。这是量化器的默认量化策略。
- `pof2s_tqt`：2 的幂值比例量化，含经训练的阈值，主要用于 DPU 中的 QAT。
- `fs`：浮点比例量化，主要用于支持浮点计算的器件，例如，CPU/GPU。
- `fsx`：经训练的量化阈值，适用于 2 的幂值比例量化，主要用于 DPU QAT。

您可通过在 `VitisQuantizer` 的构造函数中分配 `the quantize_strategy` 实参，在各内置量化策略之间进行切换。以下提供了 2 种简便的方法来配置量化策略：

#### 由 `VitisQuantizer.quantize_model()` 中的 `kwargs` 进行配置

此简便方法可供您用于覆盖默认流水线配置或者对量化运算进行全局修改。`kwargs` 是 dict 对象，其键与 JSON 文件的量化配置相匹配。如需了解有关可用的键的更多信息，请参阅 vitis_quantize.VitisQuantizer.quantize_model。

以下代码示例展示了其使用方法：

```
model = tf.keras.models.load_model('float_model.h5')
from tensorflow_model_optimization.quantization.keras import vitis_quantize
quantizer = vitis_quantize.VitisQuantizer(model)
quantizer.quantize_model(calib_dataset,
                         input_layers=['conv2'],
                         bias_bit=32,
                         activation_bit=32,
                         weight_per_channel=True)
```

本例中的量化器设置为只对模型的一部分进行量化。`conv2` 之前的层既未优化，也未量化。此外，所有激活和偏移都量化为 32 位，而不是 8 位，而且逐通道量化应用于所有权重。

#### 由 `VitisQuantizer.set_quantize_strategy()` 进行配置

对于希望全权掌控量化工具的高级用户，此 API 支持您设置新的量化策略 JSON 文件。您可以将当前配置转储到 JSON 文件中，并按需进行修改。它支持您改写默认配置、创建更高精度的量化器设置、甚至可以扩展量化配置以包含更多层类型以供量化。完成修改后，您即可给量化器设置新的 JSON 文件，以应用这些自定义配置。

以下代码示例展示了其使用方法：

```
quantizer = VitisQuantizer(model)
# Dump the current quantize strategy
quantizer.dump_quantize_strategy(dump_file='my_quantize_strategy.json',
verbose=0)

# Make modifications of the dumped file 'my_quantize_strategy.json'
# Then, set the modified json to the quantizer and do quantization
quantizer.set_quantize_strategy(new_quantize_strategy='my_quantize_strategy.
json')
quantizer.quantize_model(calib_dataset)
```

**注释：**`verbose` 是 `int` 类型的实参，用于控制已转储的 JSON 文件的详细程度。详细程度值越大，转储的量化策略就越详细。将 verbose 设置为大于或等于 2 的值即可转储完整量化策略。

## 利用浮点比例进行量化

DPU 量化使用 2 的幂值比例、对称和逐张量量化器，且需特殊处理才能进行 DPU 行为仿真。但对于支持浮点比例的其他器件，则需要采用不同的量化策略，因此引入了浮点比例量化。

- `fs` 量化策略：对 `Conv2D`、`DepthwiseConv2D`、`Conv2DTranspose` 和 `Dense` 层的输入和权重执行量化。默认不执行 Conv-BN 折叠。
- `fsx` 量化策略：相较于 `fs` 量化策略，该策略能对更多类型的层执行量化，例如，`Add`、`MaxPooling2D` 和 `AveragePooling2D`。此外，量化进程会扩展至 `Conv2D`、`DepthwiseConv2D`、`Conv2DTranspose` 和 `Dense` 层的偏差和激活。它默认包含 Conv-BN 折叠。

**注释：**`fs` 和 `fsx` 策略都是专为具有浮点支持的目标器件设计的。DPU 目前不支持浮点，因此以这些量化策略进行量化的模型无法部署到 DPU。

您可在 `VitisQuantizer` 的构造函数中将 `quantize_strategy` 设为 `fs` 或 `fsx` 来切换为使用浮点比例量化。代码示例如下所示：

```
model = tf.keras.models.load_model('float_model.h5')
from tensorflow_model_optimization.quantization.keras import vitis_quantize
quantizer = vitis_quantize.VitisQuantizer(model, quantize_strategy='fs')
quantized_model = quantizer.quantize_model(calib_dataset=calib_dataset,
                                           calib_step=100,
                                           calib_batch_size=10,
                                           **kwargs)
```

- calib_dataset：`calib_dataset` 是用作校准的代表性校准数据集。您可将 `eval_dataset`、`train_dataset` 或其他数据集作为整体来使用，或者仅使用其中一部分。
- calib_steps：`calib_steps` 表示校准步骤总数。其默认值为 None。如果 `calib_dataset` 为 `tf.data dataset`、生成器或 `keras.utils.Sequence` 实例且 steps 为 None，校准会运行到数据集耗尽为止。阵列输入不支持此实参。
- calib_batch_size：`calib_batch_size` 表示用于校准的每批次样本数。如果 `calib_dataset` 为数据集、生成器或 `keras.utils.Sequence` 实例形式，则批次大小由数据集本身控制。如果 `calib_dataset` 为 `numpy.array` 对象形式，那么默认批次大小为 32。
- **kwargs：`**kwargs` 是用户定义的量化策略配置的词典。它支持用户改写默认内置量化策略。例如，设置 bias_bit=16 即可支持该工具使用 16 位量化器来量化所有偏差。如需了解有关用户定义的配置的更多信息，请参阅 vai_q_tensorflow2 用法 部分。

## 转换为 Float16 或 BFloat16

vai_q_tensorflow2 支持为浮点模型进行数据类型转换，受支持的模型包括 Float16、BFloat16、Float 和 Double。以下代码显示了如何使用 vai_q_tensorflow2 API 执行数据类型转换。

```
model = tf.keras.models.load_model('float_model.h5')
from tensorflow_model_optimization.quantization.keras import vitis_quantize
quantizer = vitis_quantize.VitisQuantizer(model)
quantized_model = quantizer.quantize_model(convert_datatype='float16'
                                             **kwargs)
```

- convert_datatype：string。用于指示浮点模型的目标数据类型。选项包括：float16、bfloat16、float32 和 float64。默认值为 float16。

# vai_q_tensorflow2 量化感知训练

一般，量化可能会导致模型精度略有下降。但对于 MobileNets 之类的特定网络，精度损失可能会更大。为解决这一问题，量化感知训练 (QAT) 提供了一种进一步提高量化模型精度的解决方案。

QAT 与训练/微调浮点模型类似，不过在训练开始前，vai_q_tensorflow2 会重写浮点计算图，将其转换为量化模型。您可在此处找到完整示例。

QAT 的典型工作流程如下所述：

1. 准备浮点模型、数据集和训练脚本：

   开始 QAT 前，准备下列文件：

   表 16：**vai_q_tensorflow2 QAT 的输入文件**

| 编号 | 名称 | 描述 |
|---|---|---|
| 1 | 浮点模型 | 作为起点的浮点模型文件。如果您从头开始训练，可以忽略此步骤。 |
| 2 | 数据集 | 含标记的训练数据集。 |
| 3 | 训练脚本 | 用于运行模型训练/微调的 Python 脚本。 |

2. （可选）评估浮点模型。

   在执行 QAT 之前，首先评估浮点模型，以检查脚本和数据集是否精确。浮点检查点的精度和损失值也可作为 QAT 的基线。

3. 修改训练脚本并运行 QAT。

   使用 vai_q_tensorflow2 API `VitisQuantizer.get_qat_model` 将模型转换为量化模型，然后利用它进行训练/微调。下面给出了 1 个示例：

```
model = tf.keras.models.load_model('float_model.h5')


# *Call Vai_q_tensorflow2 api to create the quantize training model
from tensorflow_model_optimization.quantization.keras import
vitis_quantize
quantizer = vitis_quantize.VitisQuantizer(model)
qat_model = quantizer.get_qat_model(
    init_quant=True, # Do init PTQ quantization will help us to get a
better initial state for the quantizers, especially for the `pof2s_tqt`
```

```
strategy. Must be used together with calib_dataset
    calib_dataset=calib_dataset)

# Then run the training process with this qat_model to get the quantize
finetuned model.
# Compile the model
qat_model.compile(
        optimizer= RMSprop(learning_rate=lr_schedule),
        loss=tf.keras.losses.SparseCategoricalCrossentropy(),
        metrics=keras.metrics.SparseTopKCategoricalAccuracy())


# Start the training/finetuning
qat_model.fit(train_dataset)
```

**注释：**Vitis AI 从 2.0 版本起支持 pof2s_tqt 量化策略。它在量化器中使用经过训练的阈值，可能会改善 QAT 的结果。默认使用 Straight-Through-Estimator。8bit_tqt 方法只能用于 QAT，并且应设置 `'init_quant=True'` 以获得最佳性能。利用 PTQ 量化进行初始化可以为量化器参数生成更好的初始状态，对于 pof2s_tqt 尤其如此。否则，训练可能不会收敛。

4. 保存模型。

调用 `model.save()` 以保存训练模型，或者在 `model.fit()` 中使用回调来定期保存模型。例如：

```
# save model manually
qat_model.save('trained_model.h5')

# save the model periodically during fit using callbacks
qat_model.fit(
    train_dataset,
    callbacks = [
              keras.callbacks.ModelCheckpoint(
              filepath='./quantize_train/'
              save_best_only=True,
              monitor="sparse_categorical_accuracy",
              verbose=1,
      )])
```

5. 转换为可部署量化模型。

修改经过训练/微调的模型以满足编译器要求。例如，如果 `train_with_bn` 设为 TRUE，那么训练期间批量归一层会保持不折叠状态，在部署前必须折叠。某些量化器参数在训练期间可能会改变，超过编译器限制的范围。部署之前必须纠正这些情况。

使用 `get_deploy_model()` 函数执行这些转换，并生成可部署模型，如以下示例所示：

```
quantized_model = vitis_quantizer.get_deploy_model(qat_model)
quantized_model.save('quantized_model.h5')
```

6. （可选）评估量化模型

在 `eval_dataset` 上调用 `model.evaluate()` 以评估量化模型，就像浮点模型评估一样。

```
from tensorflow_model_optimization.quantization.keras import
vitis_quantize
quantized_model = tf.keras.models.load_model('quantized_model.h5')
```

```
quantized_model.compile(loss=tf.keras.losses.SparseCategoricalCrossentrop
y(),
        metrics= keras.metrics.SparseTopKCategoricalAccuracy())
quantized_model.evaluate(eval_dataset)
```

**建议：**使用浮点模型训练和微调，然后进行 QAT。

## 利用定制层进行量化

TensorFlow 2 提供了丰富的内置层来构造机器学习模型，还提供了直接的方法从头开始或通过组合现有层来创建特定于应用的层。`layer` 是 `tf.keras` 中的核心抽象，建议对该类采用子类化方法来开发定制层。如需了解更多详细信息，请参阅 TensorFlow 用户指南。

vai_q_tensorflow2 支持通过子类化来新建定制层，其中包括利用定制层来量化模型的功能。此外，它还提供了使用定制量化策略来量化这些定制层的实验性支持。

**注释：**在此版本中，vai_q_tensorflow2 不支持通过 `tf.keras.T` 子类化得到定制模型。请将其扁平化为层。

#### 量化具有定制层的模型

某些模型具有定制层，vai_q_tensorflow2 提供了加载定制层的接口。例如：

```
class MyCustomLayer(keras.layers.Layer):

    def __init__(self, units=32, **kwargs):
        super(MyLayer, self).__init__(kwargs)
        self.units = units


    def build(self, input_shape):
        self.w = self.add_weight(
            shape=(input_shape[-1], self.units),
            initializer="random_normal",
            trainable=True,
            name='w')
        self.b = self.add_weight(
            shape=(self.units,), initializer="zeros", trainable=True,
name='b')


    def call(self, inputs):
        return tf.matmul(inputs, self.w) + self.b


    def get_config(self):
        base_config = super(MyLayer, self).get_config()
        config = {"units": self.units}
        return dict(list(base_config.items()) + list(config.items()


# Here is a float model with custom layer," "MyCustomLayer", use the
custom_objects argument in tf.keras.models.load_model to load it.
float_model = tf.keras.models.load_model('float_model.h5',
custom_objects={'MyCustomLayer': MyCustomLayer})
```

浮点模型包含一个名为 "MyCustomLayer" 的定制层，tf.keras.model.load_model API 中的 custom_objects 实参用于加载该定制层。同样，VitisQuantizer 类提供 'custom_objects' 实参来处理定制层。下面给出了一个代码示例。custom_objects 实参是一个 dict，其中包含 {"custom_layer_class_name":"custom_layer_class"}，并以逗号来分隔多个定制层。此外，在量化含有定制层的模型时，quantize_model API 的 add_shape_info 也应设为 True，以便为这些定制层添加形状推断信息。

```
from tensorflow_model_optimization.quantization.keras import vitis_quantize
# Register the custom layer to VitisQuantizer by custom_objects argument.
quantizer = vitis_quantize.VitisQuantizer(float_model,
custom_objects={'MyCustomLayer': MyCustomLayer})
quantized_model = quantizer.quantize_model(calib_dataset=calib_dataset,
calib_step=100, calib_batch_size=10, add_shape_info=True)
```

在量化期间，这些定制层会通过 CustomLayerWrapper 进行封装并保持未量化状态。要获取完整示例，请单击此处。

**注释：**使用 dump_model API 来生成黄金结果以供在转储部署期间进行数据检查时，请设置 dump_float=True 以转储定制层的浮点权重和激活，这些层未经量化。

**（实验性）利用定制量化策略量化定制层**

默认量化策略不会对定制层进行量化，因为它们不包含在 vai_q_tensorflow2 支持的 API 列表中。但高级用户可以使用 custom_quantize_strategy 接口创建定制量化策略以开展量化实验。

定制量化策略呈现为一个 Dict 对象，其中包含 JSON 格式的量化策略项。

默认量化策略提供了量化策略格式示例，定制量化策略也遵循相同结构。但定制量化策略中的任何项都会覆盖默认策略中的对应项，而新的项则会添加到量化策略中。

您可利用此功能来量化来自上述使用定制量化策略的示例中的 MyCustomLayer 层。

```
# Define quantizer with custom quantize strategy, which quantizes w,b and
outputs 0 of MyCustomLayer objects.
my_quantize_strategy = {
    "quantize_registry_config": {
        "layer_quantize_config": [{
            "layer_type": "__main__.MyCustomLayer",
            "quantizable_weights": ["w", "b"],
            "weight_quantizers": [
                "quantizer_type":
"LastValueQuantPosQuantizer","quantizer_params": {"bit_width": 8, "method":
1, "round_mode": 0},
                "quantizer_type": "LastValueQuantPosQuantizer",
"quantizer_params": {"bit_width": 8, "method": 1, "round_mode": 0}
            ],
            "quantizable_outputs": ["0"],
            "output_quantizers": [
                "quantizer_type": "LastValueQuantPosQuantizer",
"quantizer_params": {"bit_width": 8, "method": 1, "round_mode": 1}
            ]
        }]
    }
}
quantizer = vitis_quantize.VitisQuantizer(model, custom_objects={'MyLayer':
MyLayer}, custom_quantize_strategy=my_quantize_strategy)
```

```
# The following quantizison process are all the same as before, here we do
normal PTQ as an example
quantized_model = quantizer.quantize_model(calib_dataset=calib_dataset,
calib_step=100, calib_batch_size=10)
```

## vai_q_tensorflow2 支持的运算和 API

下表列出了 vai_q_tensorflow2 支持的运算和 API。

表 17：**vai_q_tensorflow2 支持的层**

| 层类型 | 受支持的层 | 描述 |
|---|---|---|
| Core | tf.keras.layers.InputLayer | |
| Core | tf.keras.layers.Dense | |
| Core | tf.keras.layers.Activation | 如果 activation 为 relu 或 linear，则对其进行量化。<br>如果 activation 为 sigmoid 或 swish，默认情况下将被转换为 hard-sigmoid 或 hard-swish，然后被量化。<br>否则就不将其量化。 |
| Convolution | tf.keras.layers.Conv2D | |
| Convolution | tf.keras.layers.DepthwiseConv2D | |
| Convolution | tf.keras.layers.Conv2DTranspose | |
| Convolution | tf.keras.layers.SeparableConv2D | |
| Pooling | tf.keras.layers.AveragePooling2D | |
| Pooling | tf.keras.layers.MaxPooling2D | |
| Pooling | tf.keras.layers.GlobalAveragePooling | |
| Normalization | tf.keras.layers.BatchNormalization | 默认情况下，BatchNormalization 层与先前的卷积层融合。如果无法组合，则会转换为逐通道卷积。<br>在 QAT 模式下，如果 `train_with_bn` 设置为 TRUE，则 BatchNormalization 层发生伪融合。调用 `get_deploy_model` 函数时，这些层会发生融合。 |
| Regularization | tf.keras.layers.Dropout | 默认情况下，dropout 层会被移除。在 QAT 模式下，如果 tremove_dropout 设置为 FALSE，则保留 dropout 层。当调用 get_deploy_model 函数时，该层会被移除。 |
| Reshaping | tf.keras.layers.Reshape | |
| Reshaping | tf.keras.layers.Flatten | |
| Reshaping | tf.keras.UpSampling2D | |
| Reshaping | tf.keras.ZeroPadding2D | |
| Merging | tf.keras.layers.Concatenate | |
| Merging | tf.keras.layers.Add | |
| Merging | tf.keras.layers.Multiply | |
| Activation | tf.keras.layers.ReLU | |
| Activation | tf.keras.layers.Softmax | Softmax 层的输入被量化。它可以在独立 Softmax IP 上运行以获得加速。 |
| Activation | tf.keras.layers.LeakyReLU | 仅限“alpha=0”的 LeakyReLU 层。该层可量化并映射到 DPU。量化器会在内部将 alpha 值自动转换为 26/256 以匹配 DPU 的实现。含其他值的 LeakyReLU 不予量化，并将映射到 CPU。 |
| Hard_sigmoid | tf.keras.layer.ReLU(6.)(x + 3.)* (1./ 6.) | 支持的 hard_sigmoid 来自 Mobilenet_v3。<br>当前不支持 tf.keras.Activation.hard_sigmoid，且不予量化。 |

表 17：**vai_q_tensorflow2 支持的层** (续)

| 层类型 | 受支持的层 | 描述 |
|---|---|---|
| Activation | tf.keras.layers.PReLU | |

**注释：**DPU 对这些受支持的层可能存在限制。这些层可在编译期间回滚到 CPU。如需了解更多信息，请参阅 受支持的运算符和 DPU 限制。

# vai_q_tensorflow2 用法

## vitis_inspect.VitisInspector

`VitisInspector` 类的构造函数。

```
vitis_inspect.VitisInspector(
    target=None)
```

**实参**

- target：**target**：String 或 None。表示此模型部署到的目标 DPU。它可采用名称字符串 (DPUCZDX8G_ISA1_B4096)、JSON 文件路径（例如，./U50/arch.json）或指纹。默认值为 None。如果未指定目标 DPU，则会报错。

## vitis_inspect.VitisInspector.inspect_model

此函数用于执行浮点模型检查：

```
VitisInspector.inspect_model(model,
                             input_shape=None,
                             dump_model=True,
                             dump_model_file="inspect_model.h5",
                             plot=True,
                             plot_file="model.svg",
                             dump_results=True,
                             dump_results_file="inspect_results.txt",
                             verbose=0)
```

**实参**

- model：tf.keras.Model 实例。表示要检查的浮点模型。浮点模型应包含具体的输入形状。以具体的输入形状来构建浮点模型，或者以 input_shape 实参调用 inspect_model。
- input_shape：list(int) 或 list(list(int))，包含每个输入层的输入形状。如不设置，则使用输入层中的默认形状信息。使用多个输入的形状的列表，例如，inspect_model(model, input_shape=[224, 224, 3]) 或 inspect_model(model, input_shape=[[224, 224, 3], [64, 1]])。所有维度都应包含具体的值，且 batch_size 维度应省略。默认值为 None。
- dump_model：bool。指示是否转储检查的模型并将模型保存到磁盘。默认值为 False。
- dump_model_file：string。表示检查的模型文件的路径。默认值为“inspect_model.h5”。

- plot：bool。表示是否绘制 graphviz 的模型检查结果并将图像保存到磁盘。如需将模型检查结果可视化，并提供某些修改提示，那么此项很有用。仅部分输出文件类型能显示提示，例如，.svg。默认值为 False。
- plot_file：string。表示绘制模型时的模型图像文件的文件路径。默认值为 model.svg。
- dump_results：bool。表示是否转储检查结果并将文本保存到磁盘。相比于屏幕日志记录，它能将更详细的逐层结果转储到文本文件。默认值为 False。
- dump_results_file：string。表示检查结果文本文件的文件路径。默认值为“inspect_results.txt”。
- verbose：int。表示日志记录的详细程度，verbose 值越高，显示的日志记录结果越详细。默认值为 0。

## vitis_quantize.VitisQuantizer

VitisQuantizer 类的构造函数。

```
vitis_quantize.VitisQuantizer(
    float_model,
    quantize_strategy='pof2s',
    custom_quantize_strategy=None,
    custom_objects={})
```

**实参**

- float_model：tf.keras.Model 对象，包含量化配置。
- quantize_strategy：表示量化策略类型的字符串对象。可用值包括：pof2s、pof2s_tqt、fs 和 fsx。pof2s 是专为 DPU 设计的默认策略。它使用 2 的幂值比例的量化器和 Straight-Through-Estimator（直通式估算器）来执行 QAT。pof2s_tqt 是 Vitis AI 1.4 中引入的仅适用于 QAT 的策略，在量化器中使用经训练的阈值，这样可能会为 2 的幂值比例的 QAT 生成更好的结果。fs 是 Vitis AI 2.5 版本中新引入的量化策略。它会对 Conv2D、DepthwiseConv2D、Conv2DTranspose 和 Dense 层的输入和权重执行浮点比例量化。相比于 fs 量化策略，fsx 量化策略会对更多类型的层执行量化，例如，Add、MaxPooling2D 和 AveragePooling2D。此外，它还会量化偏差和激活。

  **注释：**pof2s_tqt 策略只能用于 QAT，并且应与 init_quant=True 一起使用以获得最佳性能。

  **注释：**fs 和 fsx 策略都是专为具有浮点支持的目标器件设计的。DPU 当前不支持浮点，因此以这些量化策略进行量化的模型无法部署到 DPU。
- custom_quantize_strategy：string。表示定制量化策略 JSON 文件的文件路径。
- custom_objects：dict。用于将名称（字符串）映射到定制类或函数。

## vitis_quantize.VitisQuantizer.quantize_model

此函数用于执行浮点模型训练后量化 (PTQ)，包括模型最优化、权重量化和激活训练后量化。

```
vitis_quantize.VitisQuantizer.quantize_model(
    calib_dataset=None,
    calib_batch_size=None,
    calib_steps=None,
    verbose=0,
    add_shape_info=False,
    **kwargs)
```

**实参**

- calib_dataset：`tf.data.Dataset`、`keras.utils.Sequence` 或 `np.numpy` 对象。它是用于校准的代表性数据集。您可以将 eval_dataset、train_dataset 或其他数据集整体或其中一部分用作 calib_dataset。
- calib_steps：int。表示校准步骤总数。可忽略，默认值为 None。如果 calib_dataset 为 `tf.data` 数据集、生成器或 `keras.utils.Sequence` 实例且 steps 为 None，校准会运行到数据集耗尽为止。阵列输入不支持此实参。
- calib_batch_size：int。表示用于校准的每批次样本数。如果 calib_dataset 为数据集、生成器或 `keras.utils.Sequence` 实例形式，则批次大小由数据集本身控制。如果 calib_dataset `numpy.array` 对象形式，则默认批次大小设为 32。
- verbose：int。表示日志记录的详细程度。详细程度值越大，生成的 log 日志记录越详细。默认值为 0。
- add_shape_info：bool。用于指示是否要为定制层添加形状推断信息。对于含有定制层的模型，其值必须设为 True。
- **kwargs：dict。表示用户定义的量化策略配置。它会覆盖默认内置量化策略。以下列出了详细的用户定义配置。

**`**kwargs` 中的实参**

此 API 中的 `**kwargs` 是用户定义的量化策略配置的词典。它会覆盖默认内置量化策略。例如，设置 bias_bit=16 会使该工具以 16 位量化器来量化所有偏差。以下提供了用户定义的详细配置：

- separate_conv_act：`bool` 对象，表示是否从 `Conv2D/DepthwiseConv2D/TransposeConv2D/Dense` 层分离激活函数。默认值为 `True`。
- fold_conv_bn：`bool` 对象，表示是否将 batch norm 层折叠到先前的 `Conv2D/DepthwiseConv2D/TransposeConv2D/Dense` 层中。
- convert_bn_to_dwconv：在 Vitis AI 2.0 和更低版本中命名为 `fold_bn`。`bool` 对象，表示是否将独立 BatchNormalization 层转换到 DepthwiseConv2D 层中。
- convert_sigmoid_to_hard_sigmoid：在 Vitis AI 2.0 和更低版本中命名为 `replace_sigmoid`。`bool` 对象，表示是否将 Activation(activation='sigmoid') 层和 Sigmoid 层替换为 hard sigmoid 层并执行量化。若为否，sigmoid 层将保留不予量化，并在 CPU 上进行调度。
- convert_relu_to_relu6：在 Vitis AI 2.0 和更低版本中命名为 `replace_relu6`。`bool` 对象，表示是否将 ReLU6 层替换为 ReLU 层。
- include_cle：`bool` 对象，表示是否在量化前实现跨层均衡。
- cle_steps：`int` 对象，表示用于执行跨层均衡的迭代步骤数。

- cle_to_relu6：在 Vitis AI 2.0 和更低版本中命名为 forced_cle。bool 对象，表示是否为 ReLU6 层执行强制跨层均衡。
- include_fast_ft：bool 对象，用于判定是否执行快速微调。快速微调会用校准数据集逐层调整权重，对于某些模型可获得更高的精度。默认关闭快速微调。它需要比正常 PTQ 更长的时间（不过仍然远远短于 QAT 时间，因为 calib_dataset 比训练数据集小得多）。如果遇到精度问题，请将其开启以改善性能。
- fast_ft_epochs：int 对象，表示为每层执行快速微调的迭代轮数。
- output_format：该字符串对象用于指示量化模型的保存格式。选项包括：'' 表示跳过保存；'h5' 表示保存 .h5 文件；'tf' 表示保存 saved_model 文件；'onnx' 表示保存 .onnx 文件。默认值为 ''。
- onnx_opset_version：该 int 对象表示 ONNX opset 版本。仅当 output_format 设为 'onnx' 时才生效。默认值为 11。
- output_dir：字符串对象，指示量化模型的保存目录。默认值为“./quantize_results”。
- convert_datatype：字符串对象，用于指示浮点模型的目标数据类型。选项包括：'float16'、'bfloat16'、'float32' 和 'float64'。默认值为“float16”。
- input_layers：list(string) 对象，表示要量化的 start（起始）层的名称。模型中位于这些层之前的层将不进行最优化或量化。例如，此实参可跳过某些预处理层或停止量化第一层。默认值为 []。
- output_layers：list(string) 对象，表示要量化的 end（结束）层的名称。模型中位于这些层之后的层将不进行最优化或量化。例如，此实参可跳过后处理层或停止量化最后一层。默认值为 []。
- ignore_layers：List(string) 对象，表示量化期间忽略的层的名称。例如，此实参可用于跳过某些敏感层的量化以改善准确性。默认值为 []。
- input_bit：int 对象，所有输入的位宽。默认值为 8。
- input_method：int 对象，表示在量化所有输入的过程中，用于计算比例因子的方法。选项包括：0 表示 Non_Overflow，1 表示 Min_MSE，2 表示 Min_KL，3 表示 Percentile。默认值为 0。
- input_symmetry：bool 对象，表示针对所有输入，是执行对称量化还是非对称量化。默认值为 True。
- input_per_channel：bool 对象，表示针对所有输入，是执行逐通道量化还是执行逐张量量化。默认值为 False。
- input_round_mode：int 对象，表示用于量化所有输入的舍入模式。选项包括：0 表示 HALF_TO_EVEN，1 表示 HALF_UP，2 表示 HALF_AWAY_FROM_ZERO。默认值为 1。
- input_unsigned：bool 对象，表示是否为所有输入使用无符号的整数量化。它通常用于非负数输入（范围介于 0 到 1 之间），前提是 input_unsigned 为 true。默认值为 False。
- weight_bit：int 对象，表示所有权重的位宽。默认值为 8。
- weight_method：int 对象，表示在量化所有权重的过程中，用于计算比例因子的方法。选项包括：0 表示 Non_Overflow，1 表示 Min_MSE，2 表示 Min_KL，3 表示 Percentile。默认值为 1。
- weight_symmetry：bool 对象，表示针对所有权重，是执行对称量化还是非对称量化。默认值为 True。
- weight_per_channel：bool 对象，表示针对所有权重，是执行逐通道量化还是执行逐张量量化。默认值为 False。
- weight_round_mode：int 对象，表示用于量化所有权重的舍入模式。选项包括：0 表示 HALF_TO_EVEN，1 表示 HALF_UP，2 表示 HALF_AWAY_FROM_ZERO。默认值为 0。
- weight_unsigned：bool 对象，表示是否为所有权重使用无符号的整数量化。通常在 weight_symmetry 为 false 时使用该对象。默认值为 False。

- bias_bit：`int` 对象，表示所有偏差的位宽。默认值为 8。
- bias_method：`int` 对象，表示在量化所有偏差的过程中，用于计算比例因子的方法。选项包括：0 表示 `Non_Overflow`，1 表示 `Min_MSE`，2 表示 `Min_KL`，3 表示 `Percentile`。默认值为 0。
- bias_symmetry：`bool` 对象，表示针对所有偏差，是执行对称量化还是非对称量化。默认值为 `True`。
- bias_per_channel：`bool` 对象，表示针对所有偏差，是执行逐通道量化还是执行逐张量量化。默认值为 `False`。
- bias_round_mode：`int` 对象，表示用于量化所有偏差的舍入模式。选项包括：0 表示 `HALF_TO_EVEN`，1 表示 `HALF_UP`，2 表示 `HALF_AWAY_FROM_ZERO`。默认值为 0。
- bias_unsigned：`bool` 对象，表示是否为所有偏差使用无符号的整数量化。通常在 bias_symmetry 为 false 时使用该对象。默认值为 `False`。
- activation_bit：`int` 对象，表示所有激活的位宽。默认值为 8。
- activation_method：`int` 对象，表示在量化所有激活的过程中，用于计算比例因子的方法。选项包括：0 表示 `Non_Overflow`，1 表示 `Min_MSE`，2 表示 `Min_KL`，3 表示 `Percentile`。默认值为 1。
- activation_symmetry：`bool` 对象，表示针对所有激活，是执行对称量化还是非对称量化。默认值为 `True`。
- activation_per_channel：`bool` 对象，表示针对所有激活，是执行逐通道量化还是执行逐张量量化。默认值为 `False`。
- activation_round_mode：`int` 对象，表示用于量化所有激活的舍入模式。选项包括：0 表示 `HALF_TO_EVEN`，1 表示 `HALF_UP`，2 表示 `HALF_AWAY_FROM_ZERO`。默认值为 1。
- activation_unsigned：`bool` 对象，表示是否为所有激活使用无符号的整数量化。它通常用于非负数激活（例如，ReLU 或 ReLU6），前提是 activation_symmetry 为 true。默认值为 `False`。

## vitis_quantize.VitisQuantizer.dump_model

此函数可转储量化模型的仿真结果，包括权重和激活结果：

```
vitis_quantize.VitisQuantizer.dump_model(
    model,
    dataset=None,
    output_dir='./dump_results',
    dump_float=False,
    weights_only=False)
```

**实参**

- model：`tf.keras.Model` 对象。要转储的量化模型。
- dataset：`tf.data.Dataset`、`keras.utils.Sequence` 或 `np.numpy` 对象。该数据集用于转储，如果 weights_only 设置为 `True`，则无需使用。
- output_dir：`string` 对象。表示转储结果的保存目录。
- weights_only：`bool` 对象。设置为 `True` 表示仅转储权重。将其设置为 `False` 则一并转储激活结果。

## vitis_quantize.VitisQuantizer.dump_quantize_strategy

此函数用于将当前量化策略配置转储至 JSON 文件：

```
vitis_quantize.VitisQuantizer.dump_quantize_strategy(
    dump_file='quantize_strategy.json',
    verbose=0)
```

**实参**

- dump_file：`string` 对象。表示转储的量化策略 JSON 文件的文件路径。
- verbose：`int` 对象。表示已转储的 JSON 文件的详细程度。详细程度值越大，转储的量化策略就越详细。将 verbose 设置为大于或等于 2 的值即可转储完整量化策略。默认值为 0。

## vitis_quantize.VitisQuantizer.set_quantize_strategy

此函数用于以 JSON 文件中的新配置来更新量化策略：

```
vitis_quantize.VitisQuantizer.set_quantize_strategy(
    new_quantize_strategy='quantize_strategy.json')
```

**实参**

- new_quantize_strategy：`string` 对象。表示新增的量化策略 JSON 文件的文件路径。

## vitis_quantize.VitisQuantizer.get_qat_model

此函数用于获取 QAT 的浮点模型：

```
vitis_quantize.VitisQuantizer.get_qat_model(
    init_quant=False,
    calib_dataset=None,
    calib_batch_size=None,
    calib_steps=None,
    train_with_bn=False,
    freeze_bn_delay=-1)
```

**实参**

- init_quant：`bool` 对象，用于通知是否在 QAT 之前运行初始量化。运行初始 PTQ 量化可为量化器参数产生更好的初始状态，对于 8bit_tqt 策略尤其如此。否则，训练可能不会收敛。
- calib_dataset：`tf.data.Dataset`、`keras.utils.Sequence` 或 `np.numpy` 对象。用于校准的代表性数据集。当 init_quant 设置为 `True` 时必须设置此参数。您可以将 eval_dataset、train_dataset 或其他数据集整体或其中一部分用作 calib_dataset。
- calib_steps：`int` 对象。表示初始 PTQ 步骤总数。可忽略，默认值为 None。如果 calib_dataset 为 `tf.data dataset`、生成器或 `keras.utils.Sequence` 实例，且 steps 为 None，校准会运行到数据集耗尽为止。阵列输入不支持此实参。

- calib_batch_size：`int` 对象。表示初始 PTQ 的每批次样本数。如果“calib_dataset”为数据集、生成器或 `keras.utils.Sequence` 实例形式，则批次大小由数据集本身控制。如果 calib_dataset 为 `numpy.array` 对象形式，则默认批次大小为 32。
- train_with_bn：`bool` 对象。指示在 QAT 期间是否保留 bn 层。如果设为 True，bn 参数会在量化感知训练期间更新，并帮助模型收敛。然后这些经过训练的 bn 层融合到 get_deploy_model() 函数中先前的类卷积层。如果浮点模型没有 bn 层，则此选项无效。默认值为 false。
- freeze_bn_delay：`int` 对象。冻结 bn 参数前的训练步骤。在延迟步骤后，模型会切换推断 bn 参数以避免训练中出现不稳定。仅当 train_with_bn 为 True 时才会生效。默认值为 -1，表示从不执行 bn 冻结。

## vitis_quantize.VitisQuantizer.get_deploy_model

此函数用于转换 QAT 模型并生成可部署模型。结果可馈送到 `vai_c_tensorflow` 编译器中。

```
vitis_quantize.VitisQuantizer.get_deploy_model(model)
```

### 实参

- model：`tf.keras.Model` 对象。要部署的 QAT 模型。

## 示例

### 量化

```
from tensorflow_model_optimization.quantization.keras import vitis_quantize
quantizer = vitis_quantize.VitisQuantizer(model)
quantized_model = quantizer.quantize_model(calib_dataset=calib_dataset)
```

### 评估量化模型

```
quantized_model.compile(loss=your_loss, metrics=your_metrics)
quantized_model.evaluate(eval_dataset)
```

### 加载量化模型

```
from tensorflow_model_optimization.quantization.keras import vitis_quantize
with vitis_quantize.quantize_scope():
    model = keras.models.load_model('./quantized_model.h5')
```

### 转储量化模型

```
from tensorflow_model_optimization.quantization.keras import vitis_quantize
with vitis_quantize.quantize_scope():
    quantized_model = keras.models.load_model('./quantized_model.h5')
    vitis_quantize.VitisQuantizer.dump_model(quantized_model, dump_dataset)
```

## vai_q_tensorflow2 错误代码

表 18：**vai_q_tensorflow2 错误代码**

| 错误描述 | 错误类型 | 原因和解决方案 |
|---|---|---|
| Quantizer_TF2_Unsupported_Layer | 不受支持的层类型 | 层并非 `tf.keras.layers.Layer` 或者该层尚未受支持。默认情况下，该层将不予量化，并且将映射到 CPU 上运行。您可以使用实验性支持以自定义量化策略来定义其量化行为。 |
| Quantizer_TF2_Unsupported_Model | 不受支持的模型类型 | 仅支持 tf.keras 顺序模型或功能模型。当前不支持子类化模型。请将其转换为顺序模型或功能模型，然后重试。 |
| Quantizer_TF2_Invalid_Input_Shape | 无效的输入形状 | input_shape 参数无效。请检查该参数并为其设置正确的值。 |
| Quantizer_TF2_Invalid_Calib_Dataset | 无效的校准数据集 | 校准数据集无效。请检查该参数并设置其值。 |
| Quantizer_TF2_Invalid_Target | 无效的目标 | target 参数无效。请检查该参数并为其设置正确的值。 |

# PyTorch 版本 (vai_q_pytorch)

## 安装 vai_q_pytorch

vai_q_pytorch 含 GPU 版本和 CPU 版本。它支持 PyTorch 1.2 到 2.0 版，但不支持 PyTorch 数据并行化。vai_q_pytorch 有两种安装方法：

**使用 Docker 容器安装**

Vitis AI 为量化工具（包括 vai_q_pytorch）提供了 Docker 容器。运行 GPU 或 CPU 容器后，激活 Vitis AI-PyTorch conda 环境。

```
conda activate vitis-ai-pytorch
```

**注释：**在某些情况下，如要在 conda 环境下安装某些包，但遇到权限问题，可基于 `vitis-ai-pytorch` 来创建独立的 conda 环境，而不是直接使用 `vitis-ai-pytorch`。AMD Model Zoo 中的 `pt_pointpillars_kitti_12000_100_10.8G_1.3` 模型即可作为此类示例。

含指定 PyTorch 版本（1.2 到 2.0）的新 conda 环境可使用 https://github.com/Xilinx/Vitis-AI/blob/v3.5/docker/common/replace_pytorch.sh 脚本进行创建。此脚本会执行下列操作：

- 从 Vitis AI-pytorch 克隆 conda 环境。
- 卸载原始 PyTorch、torchvision 和 vai_q_pytorch 包。
- 安装指定版本的 PyTorch 和 torchvision。
- 从源代码重新安装 vai_q_pytorch。

以下命令行可使用脚本创建新的 conda 环境：

```
replace_pytorch.sh new_conda_env_name
```

**注释：**运行该脚本前，请检查 `replace_pytorch.sh` 脚本中的 Python、PyTorch 和 cuda-toolkit 版本，并根据需要对其版本进行编辑。选择 PyTorch 版本并编辑命令行时，请遵循 PyTorch 官方网站上的指示信息进行操作。

**从源代码安装**

vai_q_pytorch 属于旨在作为 PyTorch 插件使用的 Python 包。它在 Vitis_AI_Quantizer 中开源。AMD 建议在 conda 环境下安装 vai_q_pytorch。为此，请执行以下步骤：

1. 在 .bashrc 中添加 CUDA_HOME 环境变量。

   如果 GPU 版本的 CUDA 库安装在 /usr/local/cuda 中，请将以下行添加到 .bashrc 中。如果 CUDA 位于其他目录中，那么请对该行进行相应更改：

   ```
   export CUDA_HOME=/usr/local/cuda
   ```

   对于 CPU 版本，请移除 .bashrc 中的所有 CUDA_HOME 环境变量设置。建议在 shell 窗口的命令行中使用以下命令将其清除：

   ```
   unset CUDA_HOME
   ```

2. 安装 PyTorch (1.2~2.0) 和 torchvision：

   以下代码使用 PyTorch 1.7.1 和 torchvision 0.8.2 作为示例。您可在 PyTorch 网站上找到有关其他版本的详细指示信息。

   ```
   pip install torch==1.7.1 torchvision==0.8.2
   ```

3. 安装其他依赖项：

   ```
   pip install -r requirements.txt
   ```

4. 安装 vai_q_pytorch：

   ```
   cd ./pytorch_binding
   python setup.py install
   ```

5. 验证安装：

   ```
   python -c "import pytorch_nndct"
   ```

**注释：**如果所安装的 PyTorch 版本为 1.4 或更高版本，请先导入 pytorch_nndct，然后在脚本中导入 torch。这是由于 1.4 版本改前的 PyTorch 中的漏洞所导致的。请参阅 PyTorch GitHub 议题 28536 和 19668 以获取详细信息：

```
import pytorch_nndct
import torch
```

## 量化前检查浮点模型

Vai_q_pytorch 提供了一个检查器函数，以帮助您诊断不同器件架构下的神经网络 (NN) 模型。检查器可以基于硬件约束来预测目标器件分配，以支持用户生成报告。生成的检查报告可用于指导您对 NN 模型进行修改或最优化，从而显著降低部署难度并缩短部署时间。建议先检查浮点模型，然后再继续对其进行量化。

本节以 `resnet18_quant.py` 为例，演示如何修改模型代码并应用该功能特性：

1. 导入 vai_q_pytorch 模块：

```
from pytorch_nndct.apis import Inspector
```

2. 创建含目标名称或指纹的检查器：

```
inspector = Inspector("0x603000b16013831") # by target fingerprint
or
inspector = Inspector("DPUCAHX8L_ISA0_SP") # by target name
```

3. 检查浮点模型：

```
input = torch.randn([batch_size, 3, 224, 224])
inspector.inspect(model, input)
```

4. 运行以下命令行以检查模型：

```
python resnet18_quant.py --quant_mode float --inspect
```

检查器会在屏幕上显示特殊消息，根据 verbose_level 设置，这些消息会包含特殊颜色和特殊关键字前缀“VAIQ_*”。

**注释：**这些消息显示在“[VAIQ_NOTE]: =>Start to inspect model...”与“[VAIQ_NOTE]: =>Finish inspecting”之间。

如果检查器成功运行，那么通常在输出目录 /quantize_result 下会生成以下 3 个文件：

```
inspect_{target}.txt: Target information and all the details of operations
in float model
inspect_{target}.svg: If image_format is not None. A visualization of the
inspection result is generated
inspect_{target}.gv: If image_format is not None. The dot source code of
the inspection result is generated
```

**注释：**

- 该检查器依赖于“xcompiler”包。在 Vitis AI Docker 中的 Vitis AI-Pytorch conda 环境内，提供了现成的 xcompiler。但如果 vai_q_pytorch 是由源代码安装的，那么它必须提前安装 xcompiler。
- 检查结果的可视化依赖于 dot 引擎。如果没有成功安装 dot，请在检查时设置 `image_format = None`。
- 如需获取更详细的指导信息，请参阅 example/jupyter_notebook/inspector/inspector_tutorial.ipynb。提前安装 Jupyter Notebook。运行下列命令：

```
jupyter notebook example/jupyter_notebook/inspector/
inspector_tutorial.ipynb
```

# 运行 vai_q_pytorch

vai_q_pytorch 旨在作为 PyTorch 插件来使用。AMD 提供了最简单的 API 以引入 FPGA 友好的量化功能。您只需添加几行代码就能让定义明确的模型获得量化模型对象。为此，请执行以下步骤：

## 为 vai_q_pytorch 准备文件

为 vai_q_pytorch 准备下列文件。

表 19：**vai_q_pytorch 的输入文件**

| 编号 | 名称 | 描述 |
|---|---|---|
| 1 | model.pth | 经预训练的 PyTorch 模型，通常为 PTH 文件。 |
| 2 | model.py | 包含浮点模型定义的 Python 脚本。 |
| 3 | 校准数据集 | 训练数据集的子集，包含 100 到 1000 张图像。 |

## 修改模型定义

要为 PyTorch 模型启用量化，需要修改模型定义以确保其满足以下条件。Vitis AI GitHub 提供了一个示例。

1. 用于量化的模型只能包含前传方法。所有其他函数（通常用作预处理和后处理运算）都应移至模型之外或置于派生类中。否则在前传量化模块时可能会出现意外行为，因为 API 会移除量化模块中的这些函数。
2. 这样此浮点模型应可通过 JIT 追踪测试。将浮点模块设置为评估状态，然后使用 `torch.jit.trace` 函数来测试浮点模型。欲知详情，请参阅 example/jupyter_notebook/jit_trace_test/jit_trace_test.ipynb。

PyTorch 中最常用的运算符在 vai_q_pytorch 中均受支持。如需了解更多信息，请转至 doc/support_op.md。

## 将 vai_q_pytorch API 添加至浮点脚本

假设有预训练的浮点模型和 Python 脚本用于在量化之前评估其准确性/MAP。在此情况下，Quantizer API 会用量化模块替代浮点模块。对于标准评估，评估函数有助于量化模块的前传。通过将 “quant_mode” 标志配置为 caliber，即可在训练后量化的评估过程中确定张量的量化步骤。校准完成后，将 quant_mode 设置为“test”以评估量化模型。

1. 导入 vai_q_pytorch 模块：

```
from pytorch_nndct.apis import torch_quantizer, dump_xmodel
```

2. 以需要输入的量化来生成量化器，并获取转换后的模型：

```
input = torch.randn([batch_size, 3, 224, 224])
quantizer = torch_quantizer(quant_mode, model, (input))
quant_model = quantizer.quant_model
```

3. 使用转换后的模型前传神经网络：

```
acc1_gen, acc5_gen, loss_gen = evaluate(quant_model, val_loader, loss_fn)
```

4. 输出量化结果并部署模型：

```
if quant_mode == 'calib':
    quantizer.export_quant_config()
if deploy:

    quantizer.export_torch_script()
    quantizer.export_onnx_model()
    quantizer.export_xmodel(deploy_check=False)
```

## 运行量化并获取结果

**注释：**“vai_q_pytorch”log 日志消息可通过特殊颜色和独特的关键字前缀“VAI_Q_*”来识别。log 日志消息类型包括“error”、“warning”和“note”。监控“vai_q_pytorch”log 日志消息对于验证流程状态至关重要。

1. 运行以下命令来量化该模型：

   ```
   python resnet18_quant.py --quant_mode calib --subset_len 200
   ```

   向前校准时，请借用浮点评估流程，以最大程度减少浮点脚本中的代码更改。如果最终遇到损失和精度消息，可将其忽略。

   在量化和评估过程中，控制迭代次数至关重要。通常，对于量化而言，使用 100 到 1000 张图像足矣，评估则需要整个确认集。您可在数据加载部分管理迭代次数。其中 `subset_len` 实参可管理用于网络前传的图像数量。如果浮点评估脚本缺少类似实参，则必须添加一个。

   成功执行量化命令后，会在输出目录 ./quantize_result 中生成两个重要文件。

   - ResNet.py：转换后的 vai_q_pytorch 格式模型。
   - Quant_info.json：张量的量化步骤。保留此文件以便评估量化模型。

2. 要评估量化模型，请运行以下命令：

   ```
   python resnet18_quant.py --quant_mode test
   ```

   成功执行命令后显示的精度对应于量化模型的精度。

3. 要生成 XMODEL 以便进行编译（以及供 ONNX 格式量化模型使用），批次大小应为 1。设置 `subset_len=1` 可避免冗余迭代，并运行以下命令：

   ```
   python resnet18_quant.py --quant_mode test --subset_len 1 --batch_size=1
   --deploy
   ```

   运行期间，跳过 log 日志中显示的损失和精度。Vitis AI 编译器的 XMODEL 文件会在输出目录 `./quantize_result` 中生成。此文件可用于部署到 FPGA。

   ```
   ResNet_int.xmodel: deployed XIR format model
   ResNet_int.onnx:   deployed onnx format model
   ResNet_int.pt:     deployed torch script format model
   ```

   **注释：**在 Vitis AI Docker 的 Vitis AI-pytorch conda 环境中，XIR 已准备就绪可供使用。但如果您从源代码安装 vai_q_pytorch，则需要提前安装 XIR。如不安装 XIR，将无法生成 XMODEL 文件，从而导致执行命令时出错。但您仍可在输出 log 日志中验证精度。

## 硬件感知量化策略

检查器会基于目标器件向神经网络中的运算符提供器件赋值，使 vai_q_pytorch 能够执行硬件感知量化。

以下是 example/resnet18_quant.py 中的代码示例：

```
quantizer = torch_quantizer(quant_mode=quant_mode,
                            module=model,
                            input_args=(input),
                            device=device,
                            quant_config_file=config_file,
                            target=target)
```

对于 example/resnet18_quant.py，用于执行硬件感知校准的命令行如下所示：

```
python resnet18_quant.py --quant_mode calib --target DPUCAHX8L_ISA0_SP
```

用于测试硬件感知量化模型精度的命令行如下所示：

```
python resnet18_quant.py --quant_mode test --target DPUCAHX8L_ISA0_SP
```

用于部署量化模型的命令行如下：

```
python resnet18_quant.py --quant_mode test --target DPUCAHX8L_ISA0_SP --
subset_len 1 --batch_size 1 --deploy
```

## 模块部分量化

如果无需对模型中的所有子模块进行量化，则可使用模块部分量化。除使用通用 vai_q_pytorch API 外，您也可使用成对 `QuantStub/DeQuantStub` 运算符来调整其大小。以下示例演示了如何量化 `subm0` 和 `subm2`，而不量化 `subm1`。

```
from pytorch_nndct.nn import QuantStub, DeQuantStub

class WholeModule(torch.nn.module):
    def __init__(self,...):
        self.subm0 = ...
        self.subm1 = ...
        self.subm2 = ...

        # define QuantStub/DeQuantStub submodules
        self.quant = QuantStub()
        self.dequant = DeQuantStub()

    def forward(self, input):
        input = self.quant(input) # begin of part to be quantized
        output0 = self.subm0(input)
        output0 = self.dequant(output0) # end of part to be quantized

        output1 = self.subm1(output0)

        output1 = self.quant(output1) # begin of part to be quantized
        output2 = self.subm2(output1)
        output2 = self.dequant(output2) # end of part to be quantized
```

## 寄存自定义运算

要将量化模型转换为 XMODEL、vai_q_pytorch 会提供修饰器来允许您将单一运算或一组运算寄存为 XIR 无法识别的自定义运算：

```
# Decorator API
def register_custom_op(op_type: str, attrs_list: Optional[List[str]] =
None):
  """The decorator is used to register the function as a custom operation.
  Args:
  op_type(str) - the operator type registered into quantizer.
  The type should not conflict with pytorch_nndct

  attrs_list(Optional[List[str]], optional) -
  the name list of attributes that define operation flavor.
  For example, Convolution operation has such attributes as padding,
dilation, stride and groups.
```

```
  The order of names in attrs_list should be consistent with that of the
arguments list.
  Default: None

  """
```

执行以下步骤：

1. 将一些运算聚合为一个函数。该函数的第一个实参名应为 ctx，其含义与 torch.autograd.Function 中的含义相同
2. 使用前述修饰器来修饰此函数。

```

from pytorch_nndct.utils import register_custom_op

@register_custom_op(op_type="MyOp", attrs_list=["scale_1", "scale_2"])
def custom_op(ctx, x: torch.Tensor, y:torch.Tensor, scale_1:float,
scale_2:float) -> torch.Tensor:
  return scale_1 * x + scale_2 * y

class MyModule(torch.nn.Module):
  def __init__(self):
  ...

  def forward(self, x, y):
          return custom_op(x, y, scale_1=2.0, scale_2=1.0)
```

限制：

1. 自定义运算中不允许循环运算。
2. 自定义运算只能返回一个值。

# vai_q_pytorch 快速微调

通常量化后存在少量精度损失，但特定网络（如 MobileNet）可能发生更为显著的精度损失。在此类情况下，您可先尝试通过快速微调来提升量化模型的精度。如果快速微调方法仍无法得到令人满意的结果，可以考虑使用量化感知训练 (QAT) 来进一步提高量化模型的精度。QAT 包括利用量化感知优化进行模型训练，以提升量化状态下的精度。

AdaQuant 算法[#unique_145/unique_145_Connect_42_note1] 使用一小部分未标记数据进行激活校准和权重微调。Vitis AI 量化器在“fast fine-tuning”下整合了将此算法。虽然快速微调速度稍慢，但能产生比训练后量化更好的性能。与量化感知训练 (QAT) 类似，每一轮快速微调都可能生成不同的结果。

快速微调并不包含模型训练，仅需有限次数的迭代即可。对于 ImageNet 数据集上的分类模型，实验中有 5120 张图像足矣。快速微调过程中使用的数据不需要注解。快速微调的主要要求是修改模型评估脚本；训练无需设置优化器。要使用快速微调，需要一个函数用于模型前传迭代，在此进程期间会调用该函数。建议使用原始推断代码进行重新校准以确保准确性。

您可在开源示例中找到完整示例。

```
# fast finetune model or load finetuned parameter before the test
  if fast_finetune == True:
      ft_loader, _ = load_data(
          subset_len=5120,
          train=False,
          batch_size=batch_size,
          sample_method='random',
          data_dir=args.data_dir,
          model_name=model_name)
```

```
        if quant_mode == 'calib':
            quantizer.fast_finetune(evaluate, (quant_model, ft_loader, 
loss_fn))
        elif quant_mode == 'test':
            quantizer.load_ft_param()
```

要对此 ResNet18 示例进行参数微调和重新校准，请运行以下命令：

```
python resnet18_quant.py --quant_mode calib --fast_finetune
```

要测试微调后的量化模型精度，请运行以下命令：

```
python resnet18_quant.py --quant_mode test --fast_finetune
```

要部署微调后的量化模型，请运行以下命令：

```
python resnet18_quant.py --quant_mode test --fast_finetune --subset_len 1 --
batch_size 1 --deploy
```

**注释：** Itay Hubara 等，《Improving Post Training Neural Quantization: Layer-wise Calibration and Integer Programming》，arXiv:2006.10518, 2020。

# 量化策略配置

Vai_q_pytorch 支持采用 JSON 格式的量化配置文件来处理多个量化策略配置。

### 用法

您只需将配置文件传递给 torch_quantizer API 即可激活自定义配置：

```
config_file = "./pytorch_quantize_config.json"
quantizer = torch_quantizer(quant_mode=quant_mode,
                            module=model,
                            input_args=(input),
                            device=device,
                           quant_config_file=config_file)
```

`./example/` 目录包含以下 3 个示例：`int_config.json`、`bfloat16_config.json` 和 `mix_precision_config.json`。要量化该模型，请使用配置文件搭配 `config_xxx_config.json` 命令：

```
python resnet18_quant.py --quant_mode calib --config_file int_config.json
python resnet18_quant.py --quant_mode test --config_file int_config.json
```

在配置文件示例中，“overall_quantizer_config”中的模型配置设置为 entropy（熵）校准方法和 per_tensor 量化：

```
"overall_quantize_config": {
  ...
  "method": "entropy",
  ...
  "per_channel": false,
  ...
},
```

tensor_quantize_config 使用 maxmin 校准方法和 per_tensor 量化来指定权重配置，这表示权重采用来自该模型配置的独特量化方法：

```
"tensor_quantize_config": {
  ...
  "weights": {
    ...
    "method": "maxmin",
    ...
    "per_channel": false,
    ...
    }
```

在 layer_quantize_config 列表中包含单层量化设置。此设置是由该层的类型决定的，并对 torch.nn.Conv2d 层应用逐通道量化。

```
"layer_quantize_config": [
  {
    "layer_type": "torch.nn.Conv2d",
    ...
    "overall_quantize_config": {
      ...
      "per_channel": false,
```

**配置**

- convert_relu6_to_relu：（全局量化器设置）表示是否将 ReLU6 转换为 ReLU。选项：True 或 False。
- include_cle：（全局量化器设置）表示是否使用跨层均衡。选项：True 或 False。
- include_bias_corr：（全局量化器设置）表示是否使用偏差纠正。选项：True 或 False
- target_device：（全局量化器设置）量化模型的目标类型。选项：DPU、CPU 和 GPU
- quantizable_data_type：（全局量化器设置）表示在模型中要量化的张量类型
- data_type：（张量量化设置）表示量化中使用的数据类型。选项：int、bfloat16、float16 和 float32
- bit_width：（张量量化设置）表示量化中使用的位宽。仅当数据类型为 int 时才适用。
- method：（张量量化设置）表示用于校准量化比例的方法。选项：Maxmin、Percentile、Entropy、MSE 和 diffs。仅当数据类型为 int 时才适用。
- round_mode：（张量量化设置）表示量化进程中使用的舍入方法。选项：half_even、half_up、half_down 和 std_round。仅当数据类型为 int 时才适用。
- symmetry：（张量量化设置）表示是否使用对称量化。选项：True 或 False。仅当数据类型为 int 时才适用。
- per_channel：（张量量化设置）表示是否使用 per_channel 量化。选项：True 或 False。仅当数据类型为 int 时才适用。
- signed：（张量量化设置）表示是否使用有符号量化。选项：True 或 False。仅当数据类型为 int 时才适用。
- narrow_range：（张量量化设置）表示针对有符号量化是否使用对称整数范围。选项：True 或 False。仅当数据类型为 int 时才适用。
- scale_type：（张量量化设置）表示量化进程中使用的比例类型。选项：Float（浮点）和 poweroftwo（2 的幂）。仅当数据类型为 int 时才适用。

- calib_statistic_method：（张量量化设置）当多批次数据存在不同比例时，此方法用于选择最优量化比例。选项：modal（模态）、max（最大值）、mean（平均值）和 median（中值）。仅当数据类型为 int 时才适用。

分层配置：量化配置遵循分层工作流程。

- 若在 torch_quantizer API 中未提供配置文件，则使用默认配置，此配置是专为 DPU 器件定制的，并使用 poweroftwo 量化方法。
- 如已提供配置文件，则模型配置（含全局量化器设置和全局张量量化设置）会覆盖默认设置。
- 如果配置文件中仅指定了模型配置，那么模型中的所有张量都将使用相同配置。
- 您可使用层配置来向特定层分配特定配置参数。

默认配置：

以下提供了默认配置的详细信息：

```
"convert_relu6_to_relu": false,
"include_cle": true,
"include_bias_corr": true,
"target_device": "DPU",
"quantizable_data_type": [
  "input",
  "weights",
  "bias",
  "activation"],
"datatype": "int",
"bit_width": 8,
"method": "diffs",
"round_mode": "std_round",
"symmetry": true,
"per_channel": false,
"signed": true,
"narrow_range": false,
"scale_type": "poweroftwo",
"calib_statistic_method": "modal"
```

模型配置：

在配置文件示例 `int_config.json` 中，针对模型内的所有张量都分配了相同的 int8 量化配置。在此类情况下，必须在 `overall_quantize_config` 关键字下指定全局量化参数，如下所示：

```
  "convert_relu6_to_relu": false,
  "include_cle": false,
  "keep_first_last_layer_accuracy": false,
  "keep_add_layer_accuracy": false,
  "include_bias_corr": false,
  "target_device": "CPU",
  "quantizable_data_type": [
    "input",
    "weights",
    "bias",
    "activation"],
"overall_quantize_config": {
    "datatype": "int",
    "bit_width": 8,
    "method": "maxmin",
    "round_mode": "half_even",
    "symmetry": true,
    "per_channel": false,
```

```
    "signed": true,
    "narrow_range": false,
    "scale_type": "float",
    "calib_statistic_method": "max"
}
```

与 `int_config.json` 相似，在 `bfloat16_config.json` 中以相同 bfloat16 量化设置来配置模型中的所有张量。在此情况下，全局量化参数中指定的唯一数据类型如下：

```
  "convert_relu6_to_relu": false,
  "convert_silu_to_hswish": false,
  "include_cle": false,
  "keep_first_last_layer_accuracy": false,
  "keep_add_layer_accuracy": false,
  "include_bias_corr": false,
  "target_device": "CPU",
  "quantizable_data_type": [
    "input",
    "weights",
    "bias",
    "activation"
  ],
  "overall_quantize_config": {
    "datatype": "bfloat16"
  }
```

（可选）模型中不同张量的量化配置均可单独设置。这些配置必须在 tensor_quantize_config 关键字下进行设置。以配置文件 `mix_precision_config.json` 为例，量化的全局数据类型为 bfloat16，并将偏差的数据类型更改为 float16。其余参数与全局设置保持一致：

```
"tensor_quantize_config": {

    "bias": {
        "datatype": "float16",
    }
}
```

层配置：
层量化配置必须整合到 layer_quantize_config 列表中。每一层的配置方法都涉及两个参数：层类型和层名称。执行层配置时，有 5 个要点需要注意：

- 每个层配置都必须采用词典格式。
- 在每个层配置中，“quantizable_data_type”和“overall_quantize_config”参数都是必需的。在“overall_quantize_config”参数中，必须包含该层的所有量化参数。
- 如果设置基于层类型，那么“layer_name”参数应为空值 (null)。
- 如果根据层名称来设置，则需执行模型校准进程。校准后，您需从 quantized_result 目录中生成的 Python 文件提取所需的层名称。此外，请确保 layer_type 参数为空。
- 与模型配置相似，层中不同张量的量化配置均可单独自定义。这些独立的张量配置应使用 tensor_quantize_config 关键字来指定。

在配置文件示例中，有两种层配置。其中一种配置基于层类型，另一种配置则基于层名称。在基于层类型的层配置中，torch.nn.Conv2d 层需设为特定量化参数。权重的“per_channel”参数应设为“true”，激活的“method”参数应设为“entropy”：

```
{
  "layer_type": "torch.nn.Conv2d",
  "layer_name": null,
  "quantizable_data_type": [
    "weights",
    "bias",
    "activation"],
  "overall_quantize_config": {
    "bit_width": 8,
    "method": "maxmin",
    "round_mode": "half_even",
    "symmetry": true,
    "per_channel": false,
    "signed": true,
    "narrow_range": false,
    "scale_type": "float",
    "calib_statistic_method": "max"
  },
  "tensor_quantize_config": {
    "weights": {
      "per_channel": true
    },
    "activation": {
      "method": "entropy"
    }
  }
}
```

在基于层名称的层配置中，名为“ResNet::ResNet/Conv2d[conv1]/input.2”的层须设为特定量化参数。该层中激活的“round_mode”设为“half_up”：

```
{
  "layer_type": null,
  "layer_name": "ResNet::ResNet/Conv2d[conv1]/input.2",
  "quantizable_data_type": [
    "weights",
    "bias",
    "activation"],
  "overall_quantize_config": {
    "bit_width": 8,
    "method": "maxmin",
    "round_mode": "half_even",
    "symmetry": true,
    "per_channel": false,
    "signed": true,
    "narrow_range": false,
    "scale_type": "float",
    "calib_statistic_method": "max"
  },
  "tensor_quantize_config": {
    "activation": {
      "round_mode": "half_up"
    }
  }
}
```

层名称 ResNet::ResNet/Conv2d[conv1]/input.2 是从代码示例的 `example/resnet18_quant.py` 生成的 `quantize_result/ResNet.py` 文件中提取的：

- 使用 `python resnet18_quant.py --subset_len 100` 命令运行此代码示例。这样即可生成 `quantize_result/ResNet.py` 文件。
- 在此文件中，首个卷积层的名称为“ResNet::ResNet/Conv2d[conv1]/input.2”。
- 如果该层设置为特定配置，请将层名称复制到量化配置文件。

```
import torch
import pytorch_nndct as py_nndct
class ResNet(torch.nn.Module):
  def __init__(self):
    super(ResNet, self).__init__()
    self.module_0 = py_nndct.nn.Input() #ResNet::input_0
    self.module_1 = py_nndct.nn.Conv2d(in_channels=3, out_channels=64,
kernel_size=[7, 7], stride=[2, 2], padding=[3, 3], dilation=[1, 1], groups=
1, bias=True) #ResNet::ResNet/Conv2d[conv1]/input.2
```

**配置限制**

由于存在 DPU 相关的设计约束，在 DPU 上使用整数量化和部署量化模型时，量化配置应满足以下限制：

```
method: diffs or maxmin
round_mode: std_round for weights, bias, and input; half_up for activation.
symmetry: true
per_channel: false
signed: true
narrow_range: true
scale_type: poweroftwo
calib_statistic_method: modal.
```

对于 CPU 和 GPU 器件，不存在类似的限制。但采用不同配置时，可能出现冲突。例如，校准方法“maxmin”、“percentile”或“entropy”不支持校准统计方法“modal”。此外，如果对称模式设为非对称，则不支持校准方法“mse”和“entropy”。如果发生配置冲突，那么量化工具会提供错误消息来通知用户。

# 使用新增数据格式

vai_q_pytorch 引入了一种全新的数据格式，称为块浮点 (block floating point, BFP)。在 BFP 中，任一块内的所有数值共享指数，该指数由块中最大的指数来确定。数字越小，其尾数越向右移以适应共享指数。

虽然您可使用 vai_q_pytorch 对量化结果进行评估，但当前没有将量化模型部署到硬件的选项。

**注释：**BFP 是一种新的数据格式，当前版本的 Vitis AI 工具链并不完全支持。

**用法**

BFP 提供了各种配置，包括位宽、块大小等。有两种开箱即用的配置类型（“mx6”和“mx9”）可供您直接使用，无需设置其配置项。要对模型进行量化，请执行以下步骤：

- 准备浮点模型和输入：

```
model = buid_your_model()
batch_size = 32
inputs = torch.randn([batch_size, 3, 224, 224], dtype=torch.float32)
```

- 量化浮点模型：

```
from pytorch_nndct import bfp
quantized_model = bfp.quantize_model(model, inputs, dtype='mx6')
```

- 确认量化后的模型：将该量化模型传递给确认函数，以便评估量化结果：

```
validate(quantized_model, data_loader)
```

### BFP API

```
bfp.quantize_model(model, inputs, dtype='mx6', config_file=None)
```

- 模型：要量化的浮点模块。
- 输入：此输入张量的形状应与要量化的浮点模块的实际输入相同，但可取随机值。
- dtype：预配置的 BFP 配置。可用值为 mx6 和 mx9。
- config_file：配置文件路径。该功能正在开发中。请使用预定义的 dtype。

## 量化模型的推断

### 以 torch 脚本格式推断量化模型

您可在 PyTorch 框架下以 TorchScript 格式（PT 文件）运行量化模型。执行推断之前，请先导入 pytorch_nndct 模块，因为它会为该模型设置量化运算符。代码示例如下所示：

```
import pytorch_nndct

# prepare input data
input = ......

quantized_model = torch.jit.load(quantized_model_path)

# feed input data to quantized model and make inference
output = quantized_model(input)
```

### 以 ONNX 脚本格式推断量化模型

您可通过 ONNX Runtime API 以 ONNX 格式运行量化模型。

对于含原生 Quantize 运算符和 DeQuantize 运算符的 ONNX 模型，您可使用 ONNX Runtime 来运行模型。代码示例如下所示：

```
import onnxruntime as ort

# prepare input data
input_data = ......

ort_sess = ort.InferenceSession(quantized_model_path)
input_name = ort_sess.get_inputs()[0].name
ort_output = ort_sess.run(None, {input_name: input_data})
```

您必须设置自定义运算符，然后为带有 VAI Quantize 和 DeQuantize 运算符的 ONNX 模型使用 onnxruntime_extensions 来运行模型。此设置可通过从 pytorch_nndct 导入的 load_vai_ops() 函数来完成。代码示例如下所示：

```
from onnxruntime_extensions import PyOrtFunction
from pytorch_nndct.apis import load_vai_ops

## Before running the ONNX model, custom ops must be set.
load_vai_ops()

# prepare input data
input = ......

# run using onnxruntime_extensions API
run_ort = PyOrtFunction.from_model(quantized_model_path)
ort_outputs = run_ort(input)
```

**以 XIR 格式推断量化模型**

目前尚无任何工具可运行 XIR 格式的量化模型。

## vai_q_pytorch QAT

假设有预定义的模型架构，请按照以下步骤，以来自 torchvision 的 ResNet18 为例，执行量化感知训练 (QAT)。完整模型定义可在此处找到。

1. 检查是否存在要量化的非模块运算。ResNet18 使用 `'+'` 来添加 2 个张量。将其替换为 `pytorch_nndct.nn.modules.functional.Add`。
2. 检查是否存在要多次调用的模块。通常，此类模块并无权重，最常见的是 `torch.nn.ReLu` 模块。定义多个此类模块，然后在单次前传中逐一调用这些模块。修改后满足要求的定义如下：

```
class BasicBlock(nn.Module):
  expansion = 1

  def __init__(self,
               inplanes,
               planes,
               stride=1,
               downsample=None,
               groups=1,
               base_width=64,
               dilation=1,
               norm_layer=None):
    super(BasicBlock, self).__init__()
    if norm_layer is None:
      norm_layer = nn.BatchNorm2d
    if groups != 1 or base_width != 64:
      raise ValueError('BasicBlock only supports groups=1 and
base_width=64')
    if dilation > 1:
      raise NotImplementedError("Dilation > 1 not supported in
BasicBlock")
    # Both self.conv1 and self.downsample layers downsample the input
when stride != 1
    self.conv1 = conv3x3(inplanes, planes, stride)
    self.bn1 = norm_layer(planes)
    self.relu1 = nn.ReLU(inplace=True)
    self.conv2 = conv3x3(planes, planes)
    self.bn2 = norm_layer(planes)
```

```
    self.downsample = downsample
    self.stride = stride

    # Use a functional module to replace '+'
    self.skip_add = functional.Add()

    # Additional defined module
    self.relu2 = nn.ReLU(inplace=True)

  def forward(self, x):
    identity = x

    out = self.conv1(x)
    out = self.bn1(out)
    out = self.relu1(out)

    out = self.conv2(out)
    out = self.bn2(out)

    if self.downsample is not None:
      identity = self.downsample(x)

    # Use function module instead of '+'
    # out += identity
    out = self.skip_add(out, identity)
    out = self.relu2(out)

    return out
```

3. 插入 `QuantStub` 和 `DeQuantStub`。

   使用 `QuantStub` 来量化网络的输入，使用 `DeQuantStub` 来反量化网络的输出。这样就会对单次前传中从 `QuantStub` 到 `DeQuantStub` 的任意子网络进行量化。允许存在多对 QuantStub-DeQuantStub：

```
class ResNet(nn.Module):

  def __init__(self,
               block,
               layers,
               num_classes=1000,
               zero_init_residual=False,
               groups=1,
               width_per_group=64,
               replace_stride_with_dilation=None,
               norm_layer=None):
    super(ResNet, self).__init__()
    if norm_layer is None:
      norm_layer = nn.BatchNorm2d
    self._norm_layer = norm_layer

    self.inplanes = 64
    self.dilation = 1
    if replace_stride_with_dilation iEachne:
      # each element in the tuple indicates if we should replace
      # the 2x2 stride with a dilated convolution instead
      replace_stride_with_dilation = [False, False, False]
    if len(replace_stride_with_dilation) != 3:
      raise ValueError(
          "replace_stride_with_dilation should be None "
          "or a 3-element tuple, got
{}".format(replace_stride_with_dilation))
    self.groups = groups
    self.base_width = width_per_group
```

```
    self.conv1 = nn.Conv2d(
        3, self.inplanes, kernel_size=7, stride=2, padding=3, bias=False)
    self.bn1 = norm_layer(self.inplanes)
    self.relu = nn.ReLU(inplace=True)
    self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
    self.layer1 = self._make_layer(block, 64, layers[0])
    self.layer2 = self._make_layer(
        block, 128, layers[1], stride=2,
dilate=replace_stride_with_dilation[0])
    self.layer3 = self._make_layer(
        block, 256, layers[2], stride=2,
dilate=replace_stride_with_dilation[1])
    self.layer4 = self._make_layer(
        block, 512, layers[3], stride=2,
dilate=replace_stride_with_dilation[2])
    self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
    self.fc = nn.Linear(512 * block.expansion, num_classes)

    self.quant_stub = nndct_nn.QuantStub()
    self.dequant_stub = nndct_nn.DeQuantStub()

    for m in self.modules():
      if isinstance(m, nn.Conv2d):
        nn.init.kaiming_normal_(m.weight, mode='fan_out',
nonlinearity='relu')
      elif isinstance(m, (nn.BatchNorm2d, nn.GroupNorm)):
        nn.init.constant_(m.weight, 1)
        nn.init.constant_(m.bias, 0)

    # Zero-initialize the last BN in each residual branch,
    # so that the residual branch starts with zeros, and each residual
block behaves like an identity.
    # This improves the model by 0.2~0.3% according to https://
arxiv.org/abs/1706.02677
    if zero_init_residual:
      for m in self.modules():
        if isinstance(m, Bottleneck):
          nn.init.constant_(m.bn3.weight, 0)
        elif isinstance(m, BasicBlock):
          nn.init.constant_(m.bn2.weight, 0)

  def forward(self, x):
    x = self.quant_stub(x)

    x = self.conv1(x)
    x = self.bn1(x)
    x = self.relu(x)
    x = self.maxpool(x)

    x = self.layer1(x)
    x = self.layer2(x)
    x = self.layer3(x)
    x = self.layer4(x)

    x = self.avgpool(x)
    x = torch.flatten(x, 1)
    x = self.fc(x)
    x = self.dequant_stub(x)
    return x
```

4. 使用 QAT API 创建量化器并训练模型：

```
def _resnet(arch, block, layers, pretrained, progress, **kwargs):
  model = ResNet(block, layers, **kwargs)
  if pretrained:
    #state_dict = load_state_dict_from_url(model_urls[arch],
progress=progress)
    state_dict = torch.load(model_urls[arch])
    model.load_state_dict(state_dict)
  return model

def resnet18(pretrained=False, progress=True, **kwargs):
  r"""ResNet-18 model from
    `"Deep Residual Learning for Image Recognition" <https://
arxiv.org/pdf/1512.03385.pdf>'_

    Args:
        pretrained (bool): If True, returns a model pre-trained on
ImageNet
        progress (bool): If True, displays a progress bar of the
download to stderr
    """
  return _resnet('resnet18', BasicBlock, [2, 2, 2, 2], pretrained,
progress,
                 **kwargs)

model = resnet18(pretrained=True)

# Generate dummy inputs.
input = torch.randn([batch_size, 3, 224, 224], dtype=torch.float32)

# Create a quantizer
from pytorch_nndct import QatProcessor
qat_processor = QatProcessor(model, inputs, bitwidth=8)
quantized_model = qat_processor.trainable_model()optimizer =
torch.optim.Adam(
        quantized_model.parameters(),
        lr,
        weight_decay=weight_decay)

# Use the optimizer to train del, just like a normal float model.
...
```

5. 获取可部署模型并进行测试。

   完成 QAT 后，将量化模型转换为可部署模型。可部署模型的精度可能与量化模型的精度略有不同：

```
output_dir = 'qat_result'
deployable_model = qat_processor.to_deployable(quantized_model,
output_dir)
validate(val_loader, deployable_model, criterion, gpu)
```

6. 从可部署模型导出 XMODEL。

   `batch size=1` 是 XMODEL 编译的必备条件：

```
# Use CPU mode to export xmodel.
deployable_model.cpu()
val_subset = torch.utils.data.Subset(val_dataset, list(range(1)))
subset_loader = torch.utils.data.DataLoader(
    val_subset,
    batch_size=1,
    shuffle=False,
```

```
    num_workers=8,
    pin_memory=True)
# Must forward deployable model at least 1 iteration with batch_size=1
for images, _ in subset_loader:
  deployable_model(images)
qat_processor.export_xmodel(output_dir)
```

## vai_q_pytorch QAT 要求

通常，量化可能会造成轻微的精度损失，但对于某些网络（如 MobileNet），精度损失可能更大。在这种情况下，建议先尝试快速微调。如果快速微调不能产生令人满意的结果，量化感知训练 (QAT) 可以进一步提高量化模型的精度。

不过，使用 QAT API 训练模型有具体的要求。所有要量化的运算都必须是 `torch.nn.Module` 对象的实例，而不是 Torch 函数或 Python 运算符。例如，在 PyTorch 中使用 `'+'` 添加两个张量很常见，但 QAT 却不支持。请改为将 `'+'` 替换为 `pytorch_nndct.nn.modules.functional.Add`。下表列出了需替换的运算。

表 20：**运算替换映射**

| 运算 | 替换 |
|---|---|
| + | pytorch_nndct.nn.modules.functional.Add |
| - | pytorch_nndct.nn.modules.functional.Sub |
| torch.add | pytorch_nndct.nn.modules.functional.Add |
| torch.sub | pytorch_nndct.nn.modules.functional.Sub |

**重要提示！**在前传中由于量化信息冲突，无法多次调用要量化的模块。

在要量化的网络的头尾使用 `pytorch_nndct.nn.QuantStub` 和 `pytorch_nndct.nn.DeQuantStub`。该网络可以是完整子网络，也可以是部分子网络。

## 有助于获取更好的训练结果的准则

以下提供了改善训练结果的一些技巧：

- 尽可能加载预训练的浮点权重作为初始值，以启动量化感知训练。虽然可以利用随机初始值从头开始训练，但这将增加训练复杂性且更耗时。
- 如果加载预训练的浮点权重，则必须对网络参数和量化器参数分别使用不同的初始学习率策略和学习率下降策略。总体上，对网络参数必须设置较小的学习率，对量化器参数则须设置较大的学习率。

```
model = qat_processor.trainable_model()
param_groups = [{
     'params': model.quantizer_parameters(),
     'lr': 1e-2,
     'name': 'quantizer'
}, {
     'params': model.non_quantizer_parameters(),
     'lr': 1e-5,
     'name': 'weight'
}]
optimizer = torch.optim.Adam(param_groups)
```

- 对于优化器的选择，请避免使用 torch.optim.SGD，因为此优化器可能阻止训练收敛。AMD 建议使用 torch.optim.Adam 或 torch.optim.RMSprop 及其变体。

# vai_q_pytorch 用法

本节介绍了用于为目标硬件实现量化并生成可部署模型的执行工具和 API。以下是 `pytorch_binding/pytorch_nndct/apis/quant_api.py` 模块中的 API：

## class torch_quantizer()

`torch_quantizer` 类用于创建量化器对象：

```
class torch_quantizer():
  def __init__(self,
               quant_mode: str, # ['calib', 'test']
               module: torch.nn.Module,
               input_args: Union[torch.Tensor, Sequence[Any]] = None,
               state_dict_file: Optional[str] = None,
               output_dir: str = "quantize_result",
               bitwidth: int = 8,
               device: torch.device = torch.device("cuda"),
               quant_config_file: Optional[str] = None,
               target: Optional[str]=None):
```

**实参**

- Quant_mode：这是 1 个整数，表示此进程将使用的量化模式。calib 值用于校准量化，而 test 值则用于评估量化模型。
- Module：要量化的浮点模块。
- Input_args：此输入张量的形状与要量化的浮点模块的实际输入相同，但可取随机数值。
- State_dict_file：用于对参数文件进行预训练的浮点模块。如果浮点模块已读入参数，则无需设置该参数。
- Output_dir：量化结果和中间文件的目录。默认值为 quantize_result。
- Bitwidth：全局量化位宽。默认值为 8。
- Device：在 GPU 或 CPU 上运行模型。
- Quant_config_file：包含量化策略配置的 JSON 文件的位置。
- Target：如果指定了目标器件，则会开启硬件感知量化。默认值为 None。

## def export_quant_config(self)

此函数用于导出与量化步骤相关的信息：

```
def export_quant_config(self):
```

## def export_xmodel(self, output_dir, deploy_check)

此函数可导出 xmodel 并转储运算符的输出数据，以便进行详细的数据比对。

```
def export_xmodel(self, output_dir, deploy_check):
```

**实参**

- Output_dir：量化结果和中间文件的目录。默认为“quantize_result”。
- Deploy_check：这些标志用于控制数据转储（用于进行详细的数据比对）。默认为 FALSE。如果设置为 TRUE，则二进制格式数据将会转储到 `output_dir/deploy_check_data_int/` 位置。

## def export_onnx_model(self, output_dir, verbose)-New

原生 ONNX 模型仅支持 INT8 量化和半偶数舍入。将模型从 Vitis AI 量化器转换为 ONNX 格式时，无法导出其他量化位和更多的舍入方法，如半向上舍入或向零舍入。为解决此问题，导出 ONNX 模型时，可使用 vai::QuantizeLinear 和 vai::DequantizeLinear are 替换对应的原生 ONNX 运算符。对于 DequantizeLinear，原生 ONNX 与 Vitis AI 之间的接口是相同的。但对于 QuantizeLinear，两者之间存在些许差异，总结要点如下：

- ONNX 具有输入列表（x、y_scale、y_zero_point）：Vitis AI 具有输入列表（x、valmin、valmax、scale、zero_point、method），其中，valmin 和 valmax 是量化间隔，例如，valmin=-128 和 valmax=127 对应 INT8 对称量化，采用下列舍入方法：半偶数舍入、半向上舍入、半向下舍入、向零舍入、远离零舍入等。
- 在以下定义中设置 native_onnx=True 即可获取原生 Quant-Dequant ONNX 模型。如不设置此项，则会接收到 Quant-Dequant ONNX 模型，以及 Vitis AI QuantizeLinear 和 DequantizeLinear 运算符。默认值为 False。

  此函数会以 ONNX 格式导出量化模型：

  ```
  def export_onnx_model(self, output_dir="quantize_result", verbose=False,
  dynamic_batch=False,                    opset_version=None,
  native_onnx=True, dump_layers=False, check_model=False, opt_graph=False):
  ```

表 21：**实参**

| 实参 | 描述 |
|---|---|
| Output_dir | 量化结果和中间文件的目录。默认值为 quantize_result。 |
| Verbose | 此标志用于控制日志记录的详细程度。 |
| Dynamic_batch | 此标志用于设置输入形状的批次大小是否为动态。默认值为 False。 |
| Opset_version | 默认 (ai.onnx) opset 版本目标。如不设置此项，将取当前 PyTorch 版本的最新稳定版本作为值。 |
| Native_onnx | 导出含原生 Quant-Dequant 运算符或定制 Quant-Dequant 运算符的 ONNX 模型。如果此项设为 True，就会收到原生 Quant-Dequant ONNX 模型。否则，则生成 VAI Quant-Dequant ONNX 模型。默认值为 True。 |
| Dump_layers | 在运行时期间转储 ONNX 模型中每个层的输出。默认值为 False。 |
| Check_model | 检查 XMODEL 与 ONNX 模型之间输出的差异。默认值为 False。 |
| Opt_graph | 最优化 ONNX 计算图。默认值为 False。 |

## def export_torch_script(self, output_dir, verbose)

此函数会以 TorchScript 格式导出量化模型：

```
def export_torch_script(self, output_dir, verbose):
```

**实参**

- Output_dir：量化结果和中间文件的目录。默认值为“quantize_result”
- Verbose：此标志用于控制是否显示详细 log 日志。

## Class Inspector

Class Inspector（类检查器）用于创建检查器对象，如下所示：

```
class Inspector():
def __init__(self, name_or_fingerprint: str):
```

**实参**

- name_or_fingerprint：指定硬件目标名称或指纹。

## def inspect(...)

该函数用于检测浮点模型：

```
def inspect(self,
            module: torch.nn.Module,
            input_args: Union[torch.Tensor, Tuple[Any]],
            device: torch.device = torch.device("cuda"),
            output_dir: str = "quantize_result",
            verbose_level: int = 1,
            image_format: Optional[str] = None):
```

**实参**

- module：要部署的浮点模块
- input_args：输入张量，形状与浮点模块的实际输入相同，但值可为随机数值
- device：在 GPU 或 CPU 上追踪模型
- output_dir：检查结果的目录
- verbose_level：控制屏幕上显示的检查结果的详细程度。默认值为 1。0：关闭打印检查结果 1：打印分配给 CPU 的运算的汇总报告 2：打印所有运算的器件分配的汇总报告
- image_format：导出可视化的检查结果。支持 SVG 和 PNG 图像格式。

# vai_q_pytorch 消息

本部分包含重要消息，可使用其消息 ID 进行搜索。每条信息都能帮助您发现自己模型部署中的问题，并提供潜在的解决方案。

## VAIQ_WARN

如果存在问题导致量化结果出现问题或不完整（请参阅消息文本获取详细信息），那么 Vai_q_pytorch 会显示警告消息。消息格式为 [VAIQ_WARN][MESSAGE_ID]: 消息文本。即使出现警告，量化进程仍可继续完成。

下表中列出了重要的警告消息：

表 22：**Vai_q_pytorch 警告消息表**

| 消息 ID | 描述 |
| --- | --- |
| QUANTIZER_TORCH_BATCHNORM_AFFINE | 解析模型时，BatchNorm OP 小户型 affine=False 已替换为 affine=True。 |
| QUANTIZER_TORCH_BITWIDTH_MISMATCH | 配置文件中的位宽设置。如果与 torch_quantizer API 中的设置冲突，则使用配置文件中的设置。 |
| QUANTIZER_TORCH_CONVERT_XMODEL | 转换为 XMODEL 的操作失败。请检查消息文本明确原因。 |
| QUANTIZER_TORCH_CUDA_UNAVAILABLE | CUDA (HIP) 不可用。将器件更改为 CPU。 |
| QUANTIZER_TORCH_DATA_PARALLEL | 不支持数据并行。在 vai_q_pytorch 中已移除封装文件 'torch.nn.DataParallel'。 |
| QUANTIZER_TORCH_DEPLOY_MODEL | 仅限量化感知训练进程才具有可部署的模型。 |
| QUANTIZER_TORCH_DEVICE_MISMATCH | 输入参数器件与量化器器件类型不匹配。 |
| QUANTIZER_TORCH_EXPORT_XMODEL | 由于某些原因，导致 XMODEL 生成失败。请参阅消息文本。 |
| QUANTIZER_TORCH_FINETUNE_IGNORED | 在测试模式下忽略快速微调函数。 |
| QUANTIZER_TORCH_FLOAT_OP | 默认情况下，vai_q_pytorch 会把列表 OP 识别为浮点运算符。 |
| QUANTIZER_TORCH_INSPECTOR_PATTERN | 编译器可能无法融合此 OP，并将其分配给 DPU。 |
| QUANTIZER_TORCH_LEAKYRELU | 将 LeakyReLU 的 negative_slope 强制更改为 0.1015625，因为 DPU 仅支持该值。建议将 LeakyReLU 的所有 negative_slope 都更改为 0.1015625 并重新训练浮点模型以提升部署的模型的精度。 |
| QUANTIZER_TORCH_MATPLOTLIB | 需要 matplotlib 才能进行可视化，但未能找到。需安装 matplotlib。 |
| QUANTIZER_TORCH_MEMORY_SHORTAGE | 没有足够存储器可用于快速微调，忽略此进程。尝试使用更小的校准数据集。 |
| QUANTIZER_TORCH_NO_XIR | 无法在环境中找到 XIR 包。需安装 matplotlib。 |
| QUANTIZER_TORCH_REPLACE_RELU6 | ReLU6 已替换为 ReLU。 |
| QUANTIZER_TORCH_REPLACE_SIGMOID | Sigmoid 已替换为 Hardsigmoid。 |
| QUANTIZER_TORCH_REPLACE_SILU | SiLU 已替换为 Hardswish。 |
| QUANTIZER_TORCH_SHIFT_CHECK | 量化比例太大或太小。 |
| QUANTIZER_TORCH_TENSOR_NOT_QUANTIZED | 部分张量并未量化。请检查其特殊性。 |
| QUANTIZER_TORCH_TENSOR_TYPE_NOT_QUANTIZABLE | 节点的张量类型无法量化。仅支持 float32/double/float16 量化。 |
| QUANTIZER_TORCH_TENSOR_VALUE_INVALID | 张量具有“inf”或“nan”值。将忽略此张量的量化。 |
| QUANTIZER_TORCH_TORCH_VERSION | 仅支持使用 PyTorch 1.10 和更高版本导出 TorchScript。 |
| QUANTIZER_TORCH_XIR_MISMATCH | XIR 版本与当前 vai_q_pytorch 不匹配。 |
| QUANTIZER_TORCH_XMODEL_DEVICE | 如果目标器件并非 DPU，则不支持转储 XMODEL。 |
| QUANTIZER_TORCH_REUSED_MODULE | 复用模块可能会导致 QAT 的准确性降低。请确保这与您的期望相符。请参阅消息文本，查找存在问题的模块。 |
| QUANTIZER_TORCH_DEPRECATED_ARGUMENT | 无需再指定 device 实参。器件信息可从模型直接获取。 |
| QUANTIZER_TORCH_SCALE_VALUE | 导出的比例值未经训练。 |

## VAIQ_ERROR

如果存在问题导致量化结果出现问题或者不完整（请参阅消息文本获取详细信息），那么 Vai_q_PyTorch 会显示错误消息。此类信息的格式为“[VAIQ_ERROR][MESSAGE_ID]: 消息文本”

下表列出了重要的错误信息：

表 23：**Vai_q_PyTorch 错误消息表**

| 消息 ID | 描述 |
|---|---|
| QUANTIZER_TORCH_BIAS_CORRECTION | 量化结果目录中的偏差纠正文件与当前模型不匹配。 |
| QUANTIZER_TORCH_CALIB_RESULT_MISMATCH | 加载张量的量化步骤时，发现节点名称不匹配。请确保用于测试模式的 vai_q_pytorch 和 PyTorch 版本与校准（或 QAT 训练）模式中的版本相同。 |
| QUANTIZER_TORCH_EXPORT_ONNX | 量化模块基于 PyTorch 追踪模型，此模块由于 PyTorch 内部故障导致无法导出到 ONNX。PyTorch 内部故障原因列在消息文本中。可能需要调整浮点模型代码。 |
| QUANTIZER_TORCH_EXPORT_XMODEL | 无法将计算图转换为 XMODEL。需要检查消息文本中的原因。 |
| QUANTIZER_TORCH_FAST_FINETINE | 快速微调参数文件不存在。调用模型代码中的 load_ft_param 以加载该文件。 |
| QUANTIZER_TORCH_FIX_INPUT_TYPE | 导出 ONNX 格式模型时，量化 OP 的实参中的数据类型或值违规。 |
| QUANTIZER_TORCH_ILLEGAL_BITWIDTH | 张量量化配置违规。它应为整数并在消息文本中给定的范围内。 |
| QUANTIZER_TORCH_IMPORT_KERNEL | 导入 vai_q_PyTorch 库文件时出错。检查 PyTorch 版本与 vai_q_pytorch 版本 (PyTorch_nndct.__version__) 是否匹配。 |
| QUANTIZER_TORCH_NO_CALIB_RESULT | 量化结果文件不存在。请检查校准是否已完成。 |
| QUANTIZER_TORCH_NO_CALIBRATION | 量化校准未完成执行。检查是否调用模块 FORWARD 函数。在用户代码中必须显式调用 torch_quantizer.quant_model 的 forward 函数。请参阅位于以下位置的代码示例：https://github.com/Xilinx/Vitis-AI/blob/master/src/Vitis-AI-Quantizer/vai_q_PyTorch/example/resnet18_quant.py。 |
| QUANTIZER_TORCH_NO_FORWARD | 必须在导出量化结果前调用 torch_quantizer.quant_model FORWARD 函数。请参阅位于以下位置的代码示例：https://github.com/Xilinx/Vitis-AI/blob/master/src/Vitis-AI-Quantizer/vai_q_PyTorch/example/resnet18_quant.py。 |
| QUANTIZER_TORCH_OP_REGIST | OP 的类型不能多次寄存。 |
| QUANTIZER_TORCH_PYTORCH_TRACE | 无法从模型实参和输入实参获取 PyTorch 追踪计算图。在消息文本中报告了 PyTorch 内部故障原因。可能需要调整浮点模型代码。 |
| QUANTIZER_TORCH_QUANT_CONFIG | 量化配置项违规。请参阅消息文本。 |
| QUANTIZER_TORCH_SHAPE_MISMATCH | 张量形状不匹配。请参阅消息文本。 |
| QUANTIZER_TORCH_TORCH_VERSION | Pytorch 版本针对该函数不受支持或者与 vai_q_PyTorch 版本 (PyTorch_nndct.__version__) 不匹配。请参阅消息文本。 |
| QUANTIZER_TORCH_XMODEL_BATCHSIZE | 导出 XMODEL 时，批次大小必须为 1。 |
| QUANTIZER_TORCH_INSPECTOR_OUTPUT_FORMAT | 检查器仅支持转储 SVG 格式或 PNG 格式。 |
| QUANTIZER_TORCH_INSPECTOR_INPUT_FORMAT | 检查器不再支持指纹。请改为提供架构名称。 |
| QUANTIZER_TORCH_UNSUPPORTED_OPS | 不支持 op 的量化。 |
| QUANTIZER_TORCH_TRACED_NOT_SUPPORT | 由 'torch.jit.script' 生成的模型在 vai_q_PyTorch 中不受支持。 |
| QUANTIZER_TORCH_NO_SCRIPT_MODEL | vai_q_PyTorch 找不到任何脚本模型。 |
| QUANTIZER_TORCH_REUSED_MODULE | 在前传中已多次调用量化模块。如果要在多次调用中共享量化参数，请调用 trainable_model 并设置“allow_reused_module=True”。 |

表 23：**Vai_q_PyTorch 错误消息表** (续)

| 消息 ID | 描述 |
|---|---|
| QUANTIZER_TORCH_DATA_PARALLEL_NOT_ALLOWED | 不允许 torch.nn.DataParallel 对象。 |
| QUANTIZER_TORCH_INPUT_NOT_QUANTIZED | 输入未量化。请使用 QuantStub/DeQuantStub 来定义量化作用域。 |
| QUANTIZER_TORCH_NOT_A_MODULE | 量化运算必须是“torch.nn.Module”的实例。请将该函数替换为“torch.nn.Module”对象。在消息文本中指出了原始的源码范围。 |
| QUANTIZER_TORCH_QAT_PROCESS_ERROR | 必须先调用“trainable_model”，然后再获取可部署的模型。 |
| QUANTIZER_TORCH_QAT_DEPLOYABLE_MODEL_ERROR | 给定的训练后模型已将 BN 融合到 CONV 且无法转换为可部署的模型。切勿调用 model.fuse_conv_bn()。 |
| QUANTIZER_TORCH_XMODEL_DEVICE | XMODEL 只能在 CPU 模式下导出。请使用 deployable_model(src_dir, used_for_XMODEL=True) 获取 CPU 模型。 |

# ONNX Runtime 版本 (vai_q_onnx)

## 安装 vai_q_onnx

您可按如下方式安装 vai_q_onnx：

**使用 Wheel Package 从源代码安装**

要构建 vai_q_onnx，请运行以下命令：

```
$ sh build.sh
$ pip install pkgs/*.whl
```

## 运行 vai_q_onnx

ONNX Runtime 上的量化会引用 ONNX 模型的线性量化。vai_q_onnx 工具是作为 ONNX Runtime 的插件来开发的，支持使用更多训练后量化 (PTQ) 函数来量化深度学习模型。

训练后量化 (PTQ) 是一种将预训练的浮点模型转换为量化模型的技术，模型精度损失极小。

它需要一个代表性数据集来对浮点模型运行几批次的推断，以获得每个层分布，此进程称为训练后量化。

## 准备浮点模型和校准集

运行 `vai_q_onnx` 前，请确保准备好浮点模型和校准集，包括下表中列出的文件。

表 24：**vai_q_onnx 的输入文件**

| 编号 | 名称 | 描述 |
|---|---|---|
| 1 | 浮点模型 | ONNX 格式的浮点模型。 |

表 24：**vai_q_onnx** 的输入文件 (续)

| 编号 | 名称 | 描述 |
|---|---|---|
| 2 | 校准数据集 | 训练数据集或确认数据集的子集，用于表示输入数据分发。通常，100 到 1000 张图像足以满足此处需求。 |

## （可选）浮点模型预处理

对 float32 模型进行预处理即对该模型进行变换，使其做好量化准备。其中包含以下 3 个可选步骤：

- 符号形状推断：此步骤最适合 Transformer 模型。
- 模型最优化：此步骤使用 ONNX Runtime 原生库来重写计算图，包括合并计算节点和消除冗余以改善运行时效率。
- ONNX 形状推断。

这些步骤的主对象是增强量化质量。张量形状已知时，ONNX Runtime 量化工具执行性能最佳。符号形状推断和 ONNX 形状推断对于明确张量形状都同样至关重要。符号形状推断对于基于 Transformer 的模型特别有效，而 ONNX 形状推断则适合其他模型。

模型最优化会执行某些运算符融合，以简化量化工具的工作。例如，在最优化期间，Convolution 运算符后接 BatchNormalization 可融合为一，从而实现有效量化。

ONNX Runtime 存在如下已知问题：模型最优化无法输出大小超过 2 GB 的模型。因此，对于大型模型，必须跳过最优化。

预处理 API 可在 quant_pre_process() 函数的 onnxruntime.quantization.shape_inference Python 模块中找到：

```
from onnxruntime.quantization import shape_inference

shape_inference.quant_pre_process(
    input_model_path: str,
    output_model_path: str,
    skip_optimization: bool = False,
    skip_onnx_shape: bool = False,
    skip_symbolic_shape: bool = False,
    auto_merge: bool = False,
    int_max: int = 2**31 - 1,
    guess_output_rank: bool = False,
    verbose: int = 0,
    save_as_external_data: bool = False,
    all_tensors_to_one_file: bool = False,
    external_data_location: str = "./",
    external_data_size_threshold: int = 1024,)
```

- input_model_path：输入模型文件的路径。
- output_model_path：输出模型文件的路径。
- skip_optimization：如果此项设为 true，则跳过模型最优化步骤。对于某些模型，这可能导致 ONNX 形状推断失败。
- skip_onnx_shape：跳过 ONNX 形状推断。符号形状推断对于基于 Transformer 的模型最有效。跳过所有形状推断可能降低量化有效性，因为含有未知形状的张量无法进行量化。

- skip_symbolic_shape：跳过符号形状推断。符号形状推断对于基于 Transformer 的模型最有效。跳过所有形状推断可能降低量化有效性，因为含有未知形状的张量无法进行量化。
- auto_merge：适用于符号形状推断。发生冲突时会自动合并符号维度。
- int_max：对于符号形状推断，指定最大值，这样对于诸如 slice 等运算符，即可将整数作为无边界来处理。
- guess_output_rank：对于未知运算符，猜测输出等级与输入 0 相同。
- verbose：记录推断的详细信息。选项包括 0：关闭，1：警告，3：详细。
- save_as_external_data：将 ONNX 模型保存到外部数据。
- all_tensors_to_one_file：将所有外部数据都保存到单个文件。
- external_data_location：用于保存外部文件的文件位置。
- external_data_size_threshold：外部数据的大小阈值。

## 使用 vai_q_onnx API 执行量化

静态量化方法首先使用一组称为校准数据的输入来运行模型。在这些运行期间，会为每次激活计算量化参数。这些量化参数作为常量写入量化模型并用于所有输入。

Vai_q_onnx 量化工具已将校准方法扩展至 2 的幂值比例/浮点比例量化方法。浮点比例量化方法包括 MinMax、Entropy 和 Percentile。2 的幂值比例量化方法包括 MinMax 和 MinMSE：

```
import vai_q_onnx

vai_q_onnx.quantize_static(
    model_input,
    model_output,
    calibration_data_reader,
    quant_format=vai_q_onnx.VitisQuantFormat.FixNeuron,
    calibrate_method=vai_q_onnx.PowerOfTwoMethod.MinMSE,
    input_nodes=[],
    output_nodes=[],
    extra_options=None,)
```

- model_input：要量化的模型的文件路径。
- model_output：已量化的模型的文件路径。
- calibration_data_reader：校准数据读取器。它用于枚举校准数据并为原始模型生成输入。
- quant_format：
    - QOperator：利用已量化的运算符直接量化该模型。
    - QDQ：通过在张量上插入 QuantizeLinear/DeQuantizeLinear 来量化该模型。仅支持 8 位量化。
    - VitisQuantFormat.QDQ：通过在张量上插入 VAIQuantizeLinear/VAIDeQuantizeLinear 来量化该模型。支持更大位宽和更多配置。
    - VitisQuantFormat.FixNeuron：通过在张量上插入 FixNeuron（QuantizeLinear 与 DeQuantizeLinear 组合）来量化该模型。

- calibrate_method：对于 DPU 器件，请将 calibrate_method 设为 vai_q_onnx.PowerOfTwoMethod.NonOverflow 或 vai_q_onnx.PowerOfTwoMethod.MinMSE 以便应用 2 的幂值比例量化。PowerOfTwoMethod 当前支持两种方法：MinMSE 和 NonOverflow。默认方法是 MinMSE。
- input_nodes：list(string) 对象。要量化的起始节点的名称。模型中位于这些起始节点之前的节点不进行最优化或量化。例如，此实参可用于跳过某些预处理节点或停止量化第一个节点。默认值为 []。
- output_nodes：list(string) 对象。要量化的结束节点的名称。模型中位于这些节点之后的节点不进行最优化或量化。例如，此实参可用于跳过某些后处理节点或停止量化最后一个节点。默认值为 []。

## （可选）评估量化模型

如有脚本可用于评估浮点模型（如 AMD Model Zoo 中的模型），则可将浮点模型文件替换为量化模型用于评估。

为支持自定义 FixNeuron 运算符，应导入 vai_dquantize 模块。下面给出了 1 个示例：

```
import onnxruntime as ort
from vai_q_onnx.operators.vai_ops.qdq_ops import vai_dquantize

so = ort.SessionOptions()
so.register_custom_ops_library(_lib_path())
sess = ort.InferenceSession(dump_model, so)
input_name = sess.get_inputs()[0].name
results_outputs = sess.run(None, {input_name: input_data})
```

将浮点模型文件替换为量化模型后，即可像浮点模型一样评估量化模型。

## （可选）转储仿真结果

部署量化模型后，有时需要将 CPU 和 GPU 上的仿真结果与 DPU 上的输出值进行比对。

您可使用 vai_q_onnx 的 dump_model API 随 quantized_model 一起转储仿真结果：

```
import vai_q_onnx
# This function dumps the simulation results of the quantized model,
# including weights and activation results.
vai_q_onnx.dump_model(
    model,
    dump_data_reader=None,
    output_dir='./dump_results',
    dump_float=False)
```

- model：已量化的模型的文件路径。
- dump_data_reader：用于转储的数据读取器。它会为原始模型生成输入。
- output_dir：string。表示转储结果的保存目录。成功执行此函数后，会在 output_dir 中生成转储结果。
- dump_float：Boolean。用于判定是否转储权重和激活结果的浮点值。

**注释：**dump_data_reader 的 batch_size 应设为 1 以便进行 DPU 调试。

成功执行此命令后，会在 output_dir 中生成转储结果。每个量化节点的权重和激活结果分别以 *.bin 和 *.txt 格式保存。

如果该节点输出未经量化，例如，softmax 节点，那么浮点激活结果会以 *_float.bin 和 *_float.txt 格式来保存，前提输出是“save_float”设为 True。

下表显示了转储结果的示例。

表 25：**转储结果示例**

| 批次数量 | 是否量化 | 节点名称 | 已保存的文件 |
|---|---|---|---|
| 1 | 是 | resnet_v1_50_conv1 | {output_dir}/dump_results/quant_resnet_v1_50_conv1.bin<br>{output_dir}/dump_results/quant_resnet_v1_50_conv1.txt |
| 2 | 是 | resnet_v1_50_conv1_weights | {output_dir}/dump_results/quant_resnet_v1_50_conv1_weights.bin<br>{output_dir}/dump_results/quant_resnet_v1_50_conv1_weights.txt |
| 2 | 否 | resnet_v1_50_softmax | {output_dir}/dump_results/quant_resnet_v1_50_softmax_float.bin<br>{output_dir}/dump_results/quant_resnet_v1_50_softmax_float.txt |

## vai_q_onnx 支持的运算和 API

下表列出了 vai_q_onnx 量化支持的运算和 API。此列表中不包含的运算会在量化期间跳过，并在输出模型中保留不变。

表 26：**vai_q_onnx 支持的运算**

| 运算类型 | 描述 |
|---|---|
| Conv | |
| ConvTranspose | |
| Gemm | |
| BatchNorm | 将与先前“Conv”运算融合 |
| Add | |
| Concat | |
| Relu | |
| Reshape | |
| Transpose | |
| Squeeze | |
| Resize | |
| MaxPool | |
| GlobalAveragePool | |
| AveragePool | |
| MatMul | |
| Mul | |
| Sigmoid | |
| Softmax | |

# vai_q_onnx 用法

```
vai_q_onnx.quantize_static(
    model_input,
    model_output,
    calibration_data_reader,
    quant_format=vai_q_onnx.VitisQuantFormat.FixNeuron,
    calibrate_method=vai_q_onnx.PowerOfTwoMethod.MinMSE,
    input_nodes=[],
    output_nodes=[],
    op_types_to_quantize=None,
    per_channel=False,
    reduce_range=False,
    activation_type=QuantType.QInt8,
    weight_type=QuantType.QInt8,
    nodes_to_quantize=None,
    nodes_to_exclude=None,
    optimize_model=True,
    use_external_data_format=False,
    calibrate_method=CalibrationMethod.MinMax,
    extra_options=None)
```

**实参**

- model_input：要量化的模型的文件路径。
- model_output：已量化的模型的文件路径。
- calibration_data_reader：校准数据读取器。它用于枚举校准数据并为原始模型生成输入。要使用随机数据进行快速测试，可将 calibration_data_reader 设为 None。
- quant_format：
  - QOperator：利用已量化的运算符直接量化该模型。
  - QDQ：通过在张量上插入 QuantizeLinear/DeQuantizeLinear 来量化该模型。仅支持 8 位量化。
  - VitisQuantFormat.QDQ：通过在张量上插入 VAIQuantizeLinear/VAIDeQuantizeLinear 来量化该模型。支持更大位宽和更多配置。
  - VitisQuantFormat.FixNeuron：通过在张量上插入 FixNeuron（QuantizeLinear 与 DeQuantizeLinear 组合）来量化该模型。
- calibrate_method：对于 DPU 器件，请将 calibrate_method 设为 'vai_q_onnx.PowerOfTwoMethod.NonOverflow' 或 'vai_q_onnx.PowerOfTwoMethod.MinMSE' 以便应用 2 的幂值比例量化。PowerOfTwoMethod 当前支持两种方法：MinMSE 和 NonOverflow。默认方法是 MinMSE。
- input_nodes：list(string) 对象。要量化的起始节点的名称。模型中位于这些起始节点之前的节点不进行最优化或量化。例如，此实参可跳过某些预处理节点或停止量化首个节点。默认值为 []。
- output_nodes：list(string) 对象。要量化的结束节点的名称。模型中位于这些节点之后的节点不进行最优化或量化。例如，此实参可跳过某些后处理节点或停止量化最后一个节点。默认值为 []。
- op_types_to_quantize：指定要量化的运算符的类型，例如，指定 ['Conv'] 表示仅量化 Conv。默认情况下，它会量化所有受支持的运算符。
- per_channel：按通道量化权重。DPU 当前不支持逐通道量化，因此对于 DPU，此项必须设为 False。
- reduce_range：量化含 7 位的权重。DPU 不支持 reduce_range，因此对于 DPU，此项必须设置为 False。

- weight_type：权重的量化数据类型。对于 DPU，此项必须设置为 QuantType.QInt8。如需了解有关数据类型选择的更多详细信息，请参阅 https://onnxruntime.ai/docs/performance/quantization.html。
- nodes_to_quantize：要量化的节点的名称列表。仅当该列表为“None”时，才量化其中的节点。
- nodes_to_exclude：要排除的节点的名称列表。如果此项值为 None，那么量化时会排除此列表中的节点。
- optimize_model：模型量化前先执行最优化的操作即将被弃用。不建议执行此操作，因为最优化会更改计算图，导致难以对量化损失进行调试。
- use_external_data_format：该选项用于大小较大 (>2 GB) 的模型。默认值为 False。
- extra_options：此键值对词典适用于不同场景下的各种选项。目前使用的配对：
    - ActivationSymmetric：对用于激活的校准数据加以对称处理（默认设为 False）。如果采用 PowerOfTwoMethod calibrate_method，那么应始终将 ActivationSymmetric 设为 True。
    - WeightSymmetric：对用于权重的校准数据加以对称处理（默认值为 True）。如果采用 PowerOfTwoMethod calibrate_method，那么应始终将 WeightSymmetric 设为 True。
    - ForceQuantizeNoInputCheck：默认情况下，诸如 maxpool 和 transpose 等隐性运算符的输入如果尚未量化，那么这些运算符就不会执行量化。将此项设为 True 即可强制此类运算符始终量化输入，生成量化输出。并且，可使用 nodes_to_exclude 来按节点禁用 True 行为。
    - MatMulConstBOnly：默认值为 False，表示静态模式。如果启用此项，那么仅对含 const B 的 MatMul 执行量化。
    - AddQDQPairToWeight：默认值为 False，表示量化浮点权重，并将其馈送给单独插入的 DeQuantizeLinear 节点。如果设为 True，那么它会保留浮点权重，并将 QuantizeLinear/DeQuantizeLinear 节点插入权重。如果采用 PowerOfTwoMethod calibrate_method，那么 QDQ 应始终成对显示。因此，您需将成对 qdq 添加到权重，并且它应始终将 AddQDQPairToWeight 设为 True。

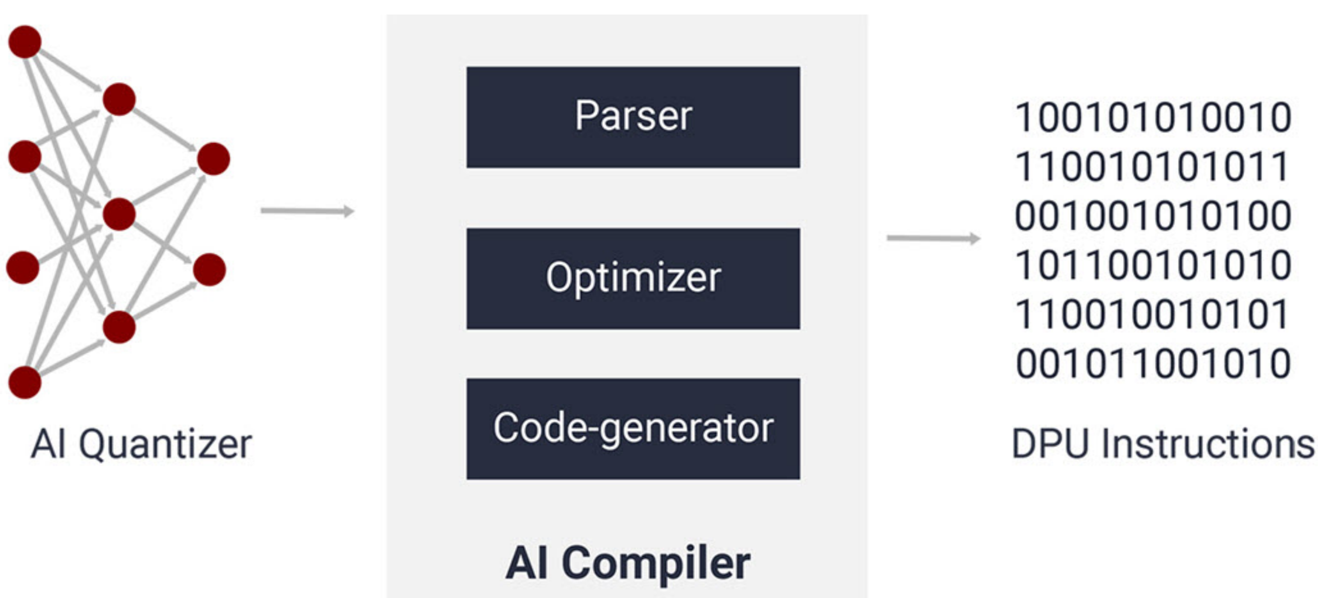
# 编译模型

## Vitis AI 编译器

AMD Vitis™ AI 编译器 (VAI_C) 是一系列编译器的统一接口，用于最优化各种深度学习处理单元 (DPU) 的神经网络计算。每个编译器都能将单个网络模型映射到单一高度优化的 DPU 指令序列。

下图显示了 VAI_C 框架的简单描述。对经过最优化和量化的输入模型的拓扑结构进行解析后，VAI_C 会构建内部计算图作为中间表示形式 (IR)，从而将其作为对应的控制流和数据流表示法。随后，它会执行多次最优化，例如，计算节点融合（例如，将批次归一化融合到主卷积）、通过利用固有并行度来保障有效的指令调度或者利用数据复用等。

图 19：**Vitis AI 编译器框架**




Vitis AI 编译器基于 DPU 微架构生成编译模型。Vitis AI 针对不同平台和应用支持多种 DPU。

表 27：**不同硬件平台上的 DPU**

| DPU 名称 | 硬件平台 |
| --- | --- |
| DPUCZDX8G | AMD Zynq™ UltraScale+™ MPSoC |
| DPUCVDX8G | AMD Versal™ 自适应 SoC VCK190 评估板，Versal AI Core 系列 |
| DPUCVDX8H | Versal 自适应 SoC VCK5000 评估套件 |
| DPUCV2DX8G | Versal 自适应 SoC VEK280 评估板，Versal AI Edge 系列，Versal 自适应 SoC V70 评估套件，Alveo V70 加速器卡 |

# 使用基于 XIR 的工具链执行编译

AMD 中间表示形式 (XIR) 是基于计算图的 AI 算法中间表示形式，专为在 FPGA 平台上执行 DPU 编译和有效部署而设计。作为高级用户，您可应用整体应用加速，将 XIR 扩展为在 Vitis AI 流程中支持自定义 IP，这样即可最大限度利用 FPGA 并发挥其全部潜能。XIR 是 Vitis AI 量化器、编译器、运行时和其他工具的基石。

## XIR

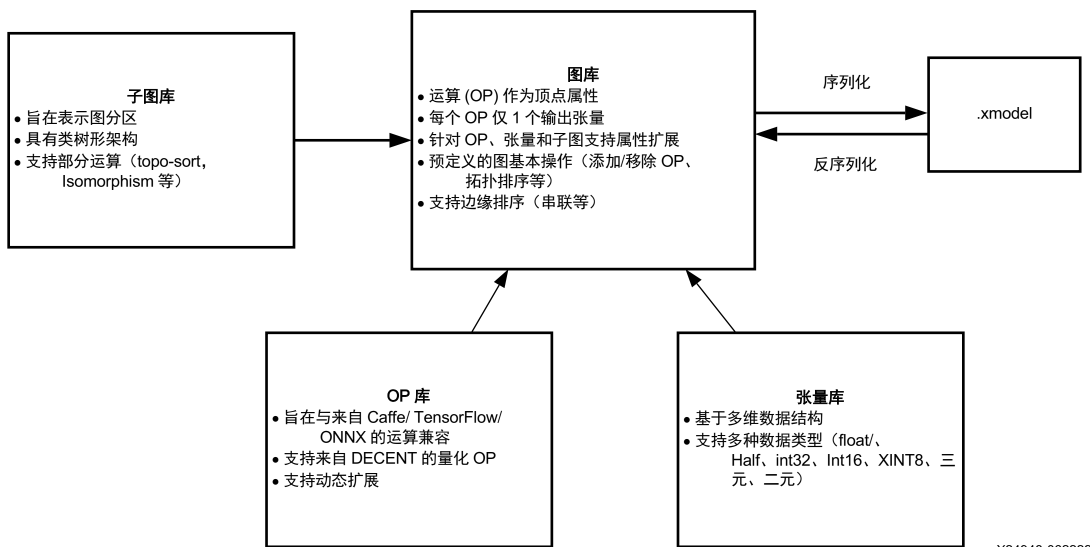
XIR 包含运算符 (Op)、张量 (Tensor)、计算图 (Graph) 和子计算图 (Subgraph) 库，可清晰灵活地呈现计算图。XIR 具有存内格式和文件格式，可支持不同用途。存内格式 XIR 为计算图对象，文件格式为 XMODEL。计算图对象可序列化为 XMODEL，而 XMODEL 亦可反序列化为计算图对象。

在 OP 中，有一组精确定义的运算符，可涵盖常用深度学习框架，例如 TensorFlow、PyTorch 和 Caffe[1]，以及所有内置 DPU 运算符。这样不仅能增强表达式的功能，还能达成核的主要目标之一：消除这些框架之间的差异，并为用户和开发者提供统一的表示法。

XIR 还提供名为 PyXIR 的 Python API，使您能够在 Python 环境中完整访问 XIR。您可凭借 PyXIR 搭配现有 XIR 工具来协同开发和集成 Python 工程。有此集成，就无需再耗费大量精力来弥合不同语言之间的差异。

1 从 Vitis AI 2.5 版起，弃用 Caffe。如需了解更多信息，请参阅《Vitis AI 2.0 用户指南》。

图 20：基于 XIR 的流程




### xir::Graph

计算图 (Graph) 是 XIR 的核心组件。它包含多个重要的 API，例如，`xir::Graph::serialize`、`xir::Graph::deserialize` 和 `xir::Graph::topological_sort`。

计算图就像是容器，可保存运算符作为其顶点，并利用“生产者 - 使用者”关系作为边缘。

### xir::Op

XIR 中的运算符是 XIR 中的运算符定义的实例或者是从 XIR 扩展的运算符定义的实例。所有运算符实例都只能由计算图根据预定义的内置/扩展运算符定义库来进行创建或添加。运算符定义主要包含输入实参和内部固有属性。

除了预定义的内部固有属性外，运算符实例也能通过应用 `xir::Op::set_attr` API 来携带更多外部非固有属性。每个运算符实例都只能包含单个输出张量，但可包含多个扇出运算符。

### xir::Tensor

张量是 XIR 中的另一个重要的类。不同于其他框架的张量定义，XIR 的张量仅描述它所表示的数据块。实际的数据块则排除在张量外。

张量的关键属性为数据类型和形状。

### xir::Subgraph

XIR 的子计算图是类树层级，将一组运算符分割为多个不重叠的集合。计算图的整个运算符集合可视作为根。子计算图可嵌套，但不得重叠。嵌套的内部子计算图必须为外部子计算图的子集。

# 面向 DPU 的编译

基于 XIR 的编译器会提取经量化的 Caffe[2]、TensorFlow、TensorFlow2.x 或 PyTorch 模型作为输入。首先，它将输入模型转换为 XIR 格式，作为后续流程的基础。不同框架间的大部分差异由此得以消除，并被转变为统一的 XIR 表示法。随后，它会对计算图应用各种最优化操作，并基于运算是否可在 DPU 上执行来将该计算图细分为多个子计算图。然后根据需要，对每个子计算图应用架构相关最优化。对于 DPU 子计算图，编译器会生成并连接到指令流。最后，将经过最优化且包含 VART 所需的必要信息和指令的计算图序列化到已编译的 XMODEL 文件中。

基于 XIR 的编译器可支持边缘 Zynq UltraScale+ MPSoC 平台上的 DPUCZDX8G 系列、Alveo 平台上的 DPUCADF8H、Alveo HBM 平台上的 DPUCAHX8H（专为高吞吐量应用而最优化）、Versal 边缘平台上的 DPUCVDX8G 和 DPUCV2DX8G 以及 Versal 数据中心平台上的 DPUCVDX8H。您可在 `/opt/vitis_ai/compiler/arch` 中找到对应这些平台的 `arch.json` 文件。

使用 VAI_C 编译 Caffe 或 TensorFlow 模型的步骤与先前 DPU 所用步骤并无不同。假定您已成功安装包含 VAI_C 的 Vitis AI 包，并已使用 vai_quantizer 压缩您的模型。

### TensorFlow

对于 TensorFlow，vai_q_tensorflow 会生成 PB 文件 (`quantize_eval_model.pb`)。vai_q_tensorflow 会生成 2 个 pb 文件。`quantize_eval_model.pb` 文件作为输入文件，供基于 XIR 的编译器使用。编译命令如下。

```
vai_c_tensorflow -f /PATH/TO/quantize_eval_model.pb -a /PATH/TO/arch.json -
o /OUTPUTPATH -n netname
```

输出与 Caffe 的输出相同。

有时，TensorFlow 模型不包含输入张量形状信息，因为它可能导致编译失败。您可使用 `--options '{"input_shape": "1,224,224,3"}'` 之类的附加选项来指定输入张量形状。

### TensorFlow 2.x

对于 TensorFlow 2.x，量化器会以 hdf5 格式生成量化模型。

```
vai_c_tensorflow2 -m /PATH/TO/quantized.h5 -a /PATH/TO/arch.json -o /
OUTPUTPATH -n netname
```

当前，vai_c_tensorflow2 仅支持 Keras 函数式 API。

### PyTorch

对于 PyTorch，量化器 NNDCT 直接以 XIR 格式输出量化模型。请使用 vai_c_xir 对其进行编译。

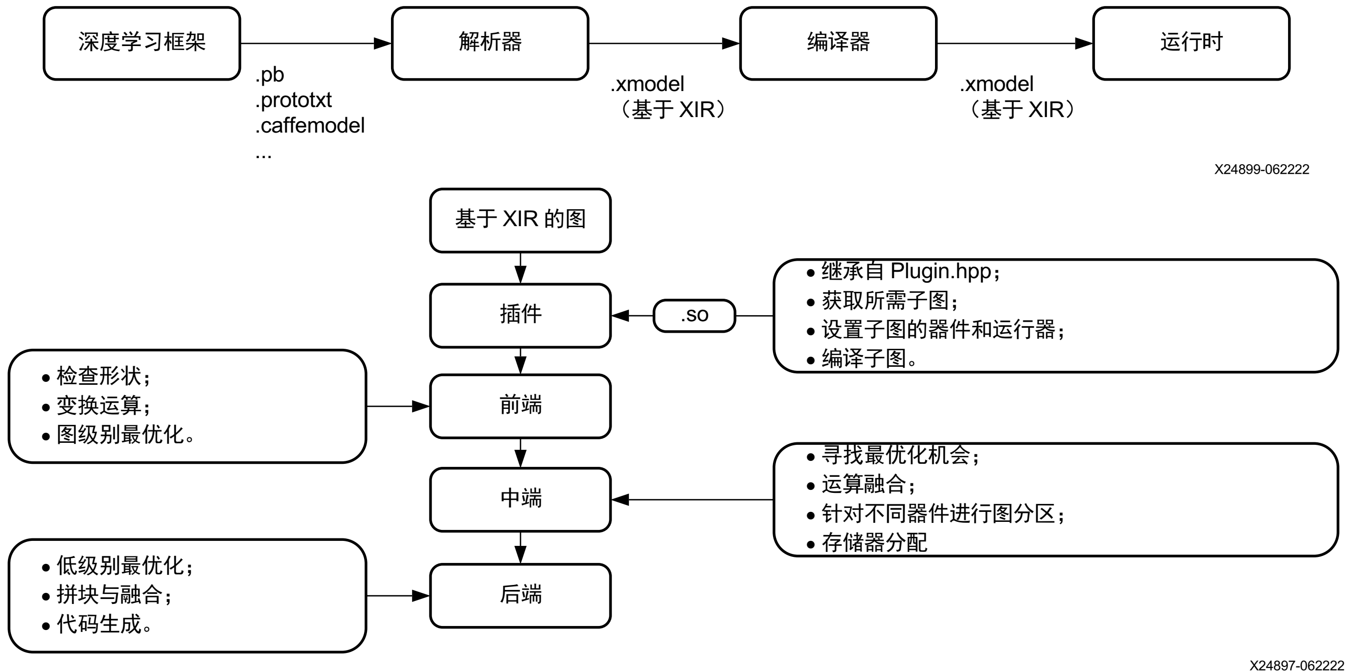
```
vai_c_xir -x /PATH/TO/quantized.xmodel -a /PATH/TO/arch.json -o /OUTPUTPATH
-n netname
```

# 针对自定义加速器进行编译

基于 XIR 的编译器可在独立于框架的 XIR 计算图的关联环境中工作，此 XIR 计算图是从深度学习框架生成的。解析器用于移除 CNN 模型中特定于框架的属性，并将这些模型转换为基于 XIR 的计算图。编译器可将计算图拆分为多个子计算图、利用异构最优化并为子计算图生成最优化的机器代码。

[2] 从 Vitis AI 2.5 版起，已弃用 Caffe。如需了解相关信息，请参阅《Vitis AI 2.0 用户指南》。

图 21：**编译流程**




如果模型包含 DPU 不支持的运算，则创建的部分子计算图将被映射到 CPU。FPGA 非常强大，可支持您创建特定 IP 以加速这些运算，从而获得更好的端到端性能。为了使用基于 XIR 的工具链来支持对 IP 进行自定义加速，请使用流水线指定的插件来扩展 XIR 和编译器。

在 `Plugin.hpp` 中，已声明推断类插件。插件将在编译器启动前按序执行，以对计算图进行编译，供 DPU 使用。首先为每个运算符创建子级子计算图，且插件会选取要加速的运算符。它会将这些运算符合并为更大的子计算图、将其映射到自定义 IP，并为运行时 (VART::Runner) 附上必要的信息（例如，有关子计算图的指令）。

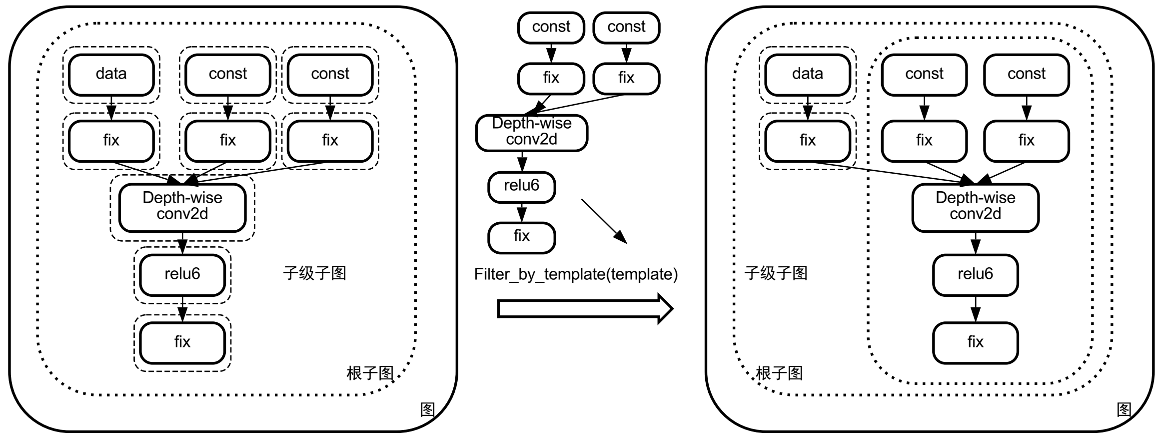
### 实现插件

1. 实现 `Plugin::partition()`

   在 `std::set<xir::Subgraph*> partition(xir::Graph* graph)` 中，使用下列帮助函数选取所需运算，并将其合并为器件级子计算图。

   - `xir::Subgraph* filter_by_name(xir::Graph* graph, const std::string& name)` 用于返回含特定名称的子计算图
   - `std::set<xir::Subgraph*> filter_by_type(xir::Graph* graph, const std::string& type)` 用于返回特定类型的子计算图。
   - `std::set<xir::Subgraph*> filter_by_template(xir::Graph* graph, xir::GraphTemplate* temp)` 用于返回含特定结构的子计算图。

图 22：**按模板筛选**




- `std::set<xir::Subgraph*> filter(xir::Graph* graph, std::function<std::set<xir::Subgraph*>(std::set<xir::Subgraph*>)> func)` 支持您按自定义函数对子计算图进行筛选。此方法可帮助您查找所有未编译的子计算图。

  如需合并子级子计算图，请使用 `merge_subgraph()` 帮助函数。但此函数只能合并相同级别的子计算图。如果子计算图列表无法合并为单一子计算图，那么帮助函数会尽可能将其合并。

2. 请在 `Plugin::partition()` 函数中为所选取的子计算图指定名称、器件和运行器。
3. 实现 `Plugin::compile(xir::Subgraph*)`。针对 partition() 函数返回的所有子计算图都会调用该函数。您可附加有关子计算图的信息以供运行时使用。

**构建插件**

创建外部 `get_plugin()` 函数，并将实现构建到共享库中。

```
extern "C" plugin* get_plugin() { return new YOURPLUGIN(); }
```

**使用插件**

您可在 vai_c 命令行选项中使用 `--options '{"plugins": "plugin0,plugin1"}'` 将插件库传递到编译器。执行插件时，编译器会打开库，并通过加载名为“get_plugin”的外部函数来创建插件实例。如果指定多个插件，则将按命令行选项定义的顺序来执行这些插件。DPU 和 CPU 编译将在所有插件都实现后再执行。

**样本**

请访问 https://github.com/Xilinx/Vitis-AI/tree/v3.5/src/vai_runtime/plugin-samples 以获取样本。

# 受支持的运算符和 DPU 限制

AMD 致力于持续不断改进 DPU IP 和编译器，以期在增加受支持的运算符的同时提升性能。下表列出了 DPU 可支持的部分典型运算和配置，例如，内核大小和步幅。如果运算配置超出这些限制，则会将运算符分配到 CPU。此外，DPU 可支持的运算符还取决于 DPU 类型、ISA 版本和配置。

您可根据自己的需求配置 DPU。您可以为 DPU 参考设计工程选择引擎、调整内部参数并创建自己的 DPU IP，但不同配置之间的限制可能存在较大差异。请使用以下产品指南了解配置信息，或者搭配您的 DPU 配置来编译模型。编译器会告诉您哪些运算符可分配给 CPU。该表显示了每个 DPU 架构的特定配置。

- 《适用于 Zynq UltraScale+ MPSoC 的 DPUCZDX8G 产品指南》(PG338)

- 《适用于卷积神经网络的 DPUCAHX8H 产品指南》(PG367)
- 《适用于 Versal 自适应 SoC 的 DPUCVDX8G 产品指南》(PG389)
- 《适用于卷积神经网络的 DPUCVDX8H v1.0 LogiCORE IP 产品指南》(PG403)
- 《适用于 Versal 自适应 SoC 的 DPUCV2DX8G 产品指南》(PG425)

以下运算符最初在不同深度学习框架内定义。编译器可自动解析这些运算符、将其转换为 XIR 格式并将其分配到 DPU 或 CPU。此外，还列出了这些运算符部分支持的工具。您可以使用 检查浮点模型 来检查模型中的运算符。

## 当前支持的运算符

表 28：当前支持的运算符

| CNN 中的典型运算类型 | 参数 | DPUCZDX8G_ISA1_B4096[3]（ZCU102 和 ZCU104） | DPUCAHX8L_ISA0（U50、U50LV 和 U280） | DPUCVDX8G_ISA3_C32B3[4](VCK190) | DPUCAHX8H_ISA2_DWC[1]（U50、U55C、U50LV 和 U280） | DPUCADF8H_ISA0（U200 和 U250） | DPUCVDX8H_ISA1_F2W4_4PE[2] (VCK5000) | DPUCV2DX8G_ISA1_C20B1 [5](VEK280/V70) |
|---|---|---|---|---|---|---|---|---|
| 内部参数 | | **channel_parallel：16**<br>bank_depth：2048<br>bank_num：8 | **channel_parallel：32**<br>bank_depth：4096 | **channel_parallel：16**<br>bank_depth：8192<br>bank_num：8 | **channel_parallel：16**<br>bank_depth：2048 | **channel_parallel：16**<br>bank_depth：8192 | **channel_parallel：64**<br>bank_depth：2048 | **channel_parallel：32**<br>bank_depth：65528<br>bank_num：1 |
| conv2d | Kernel size | w, h: [1, 16] | w, h: [1, 16] | w, h: [1, 16]<br>w * h * ceil(input_channel/ 2048) <= 64 | w, h: [1, 16] | w, h: [1, 16] | w, h: [1, 16] | w, h: [1, 16]<br>256 * h * w <= 13760 |
| | Strides | w, h: [1, 8] | w, h: [1, 4] | w, h: [1, 8] | w, h: [1, 4] | w, h: [1, 8] | w, h: [1, 4] | w, h: [1, 8] |
| | Dilation | dilation * input_channel <= 256 * channel_parallel | | | | | | |
| | Paddings | pad_left, pad_right: [0, (kernel_w - 1) * dilation_w] | | | | | | |
| | | pad_top, pad_bottom: [0, (kernel_h - 1) * dilation_h] | | | | | | |
| | In Size | kernel_w * kernel_h * ceil(input_channel / channel_parallel) <= bank_depth | | | | | | kernel_w * kernel_h * ceil(input_channel / channel_parallel) * ceil(channel_parallel / 4) + 4 <= bank_depth |
| | | input_channel <= 256 * channel_parallel | | input_channel <= 256 * channel_parallel | | | | input_channel <= 256 * channel_parallel |
| | Out Size | output_channel <= 256 * channel_parallel | | | | | | |
| | Activation | ReLU、LeakyReLU、ReLU6、Hard-Swish 和 Hard-Sigmoid | ReLU 和 ReLU6 | ReLU、LeakyReLU、ReLU6、Hard-Swish 和 Hard-Sigmoid | ReLU、LeakyReLU 和 ReLU6 | ReLU 和 LeakyReLU | ReLU、LeakyReLU、ReLU6、Hard-Swish 和 Hard-Sigmoid | ReLU、LeakyReLU、ReLU6、Hard-Swish 和 Hard-Sigmoid |
| | Group* (Caffe) | group==1 | | | | | | |

表 28：当前支持的运算符 (续)

| CNN 中的典型运算类型 | 参数 | DPUCZDX8G_ISA1_B4096[3]（ZCU102 和 ZCU104） | DPUCAHX8L_ISA0（U50、U50LV 和 U280） | DPUCVDX8G_ISA3_C32B3[4](VCK190) | DPUCAHX8H_ISA2_DWC[1]（U50、U55C、U50LV 和 U280） | DPUCADF8H_ISA0（U200 和 U250） | DPUCVDX8H_ISA1_F2W4_4PE[2] (VCK5000) | DPUCV2DX8G_ISA1_C20B1 [5](VEK280/V70) |
|---|---|---|---|---|---|---|---|---|
| 内部参数 | | **channel_parallel：16**<br>bank_depth：2048<br>bank_num：8 | **channel_parallel：32**<br>bank_depth：4096 | **channel_parallel：16**<br>bank_depth：8192<br>bank_num：8 | **channel_parallel：16**<br>bank_depth：2048 | **channel_parallel：16**<br>bank_depth：8192 | **channel_parallel：64**<br>bank_depth：2048 | **channel_parallel：32**<br>bank_depth：65528<br>bank_num：1 |
| depthwise-conv2d | Kernel size | w, h: [1, 256] | w, h: [3] | w, h: [1, 256] | w, h: {1, 2, 3, 5, 7} | 不支持 | w, h: [1, 8] | w, h: [1, 256]<br>h * w <= 431 |
| | Strides | w, h: [1, 256] | w, h: [1, 2] | w, h: [1, 256] | w, h: [1, 4] | | w, h: [1, 4] | w, h: [1, 256] |
| | dilation | dilation * input_channel <= 256 * channel_parallel | | | | | dilation * input_channel <= 256 * channel_parallel | |
| | Paddings | pad_left, pad_right: [0, min((kernel_w - 1), 15) * dilation_w] | pad_left, pad_right: [0, (kernel_w - 1) * dilation_w] | pad_left, pad_right: [0, min((kernel_w-1), 15) * dilation_w] | pad_left, pad_right: [0, (kernel_w - 1) * dilation_w] | | pad_left, pad_right: [0, (kernel_w - 1) * dilation_w] | pad_left, pad_right: [0, min((kernel_w-1), 15) * dilation_w] |
| | | pad_top, pad_bottom: [0, min((kernel_h - 1), 15) * dilation_h] | pad_top, pad_bottom: [0, (kernel_h - 1) * dilation_h] | pad_top, pad_bottom: [0, min((kernel_h-1), 15) * dilation_h] | pad_top, pad_bottom: [0, (kernel_h - 1) * dilation_h] | | pad_top, pad_bottom: [0, (kernel_h - 1) * dilation_h] | pad_top, pad_bottom: [0, min((kernel_h-1), 15) * dilation_h] |
| | In Size | kernel_w * kernel_h * ceil(input_channel / channel_parallel) <= bank_depth | | | | | kernel_w * kernel_h * ceil(input_channel / channel_parallel) <= bank_depth | (6 * stride_w + kernel_w) * kernel_h + 4 <= 512 |
| | Out Size | output_channel <= 256 * channel_parallel | | | | | output_channel <= 256 * channel_parallel | |
| | Activation | ReLU、ReLU6、LeakyReLU[6]、Hard-Swish 和 Hard-Sigmoid | ReLU 和 ReLU6 | ReLU、ReLU6、LeakyReLU[7]、Hard-Swish 和 Hard-Sigmoid | ReLU 和 ReLU6 | | ReLU 和 ReLU6 | ReLU、ReLU6、LeakyReLU、Hard-Swish 和 Hard-Sigmoid |
| | Group* (Caffe) | group==input_channel | | | | | group==input_channel | |

表 28：当前支持的运算符 (续)

| CNN 中的典型运算类型 | 参数 | DPUCZDX8G_ISA1_B4096[3]（ZCU102 和 ZCU104） | DPUCAHX8L_ISA0（U50、U50LV 和 U280） | DPUCVDX8G_ISA3_C32B3[4](VCK190) | DPUCAHX8H_ISA2_DWC[1]（U50、U55C、U50LV 和 U280） | DPUCADF8H_ISA0（U200 和 U250） | DPUCVDX8H_ISA1_F2W4_4PE[2] (VCK5000) | DPUCV2DX8G_ISA1_C20B1 [5](VEK280/V70) |
|---|---|---|---|---|---|---|---|---|
| 内部参数 | | **channel_parallel：16**<br>bank_depth：2048<br>bank_num：8 | **channel_parallel：32**<br>bank_depth：4096 | **channel_parallel：16**<br>bank_depth：8192<br>bank_num：8 | **channel_parallel：16**<br>bank_depth：2048 | **channel_parallel：16**<br>bank_depth：8192 | **channel_parallel：64**<br>bank_depth：2048 | **channel_parallel：32**<br>bank_depth：65528<br>bank_num：1 |
| transposed-conv2d | Kernel size | kernel_w/stride_w, kernel_h/stride_h: [1, 16] | | | | | | |
| | Strides | | | | | | | |
| | Paddings | pad_left, pad_right: [0, kernel_w-1] | | | | | | |
| | | pad_top, pad_bottom: [0, kernel_h-1] | | | | | | |
| | Out Size | output_channel <= 256 * channel_parallel | | | | | | |
| | Activation | ReLU、LeakyReLU、ReLU6、Hard-Swish 和 Hard-Sigmoid | ReLU 和 ReLU6 | ReLU、LeakyReLU、ReLU6、Hard-Swish 和 Hard-Sigmoid | ReLU、LeakyReLU 和 ReLU6 | ReLU 和 LeakyReLU | ReLU、LeakyReLU、ReLU6、Hard-Swish 和 Hard-Sigmoid | ReLU、LeakyReLU、ReLU6、Hard-Swish 和 Hard-Sigmoid |

表 28：当前支持的运算符 (续)

| CNN 中的典型运算类型 | 参数 | DPUCZDX8G_ISA1_B4096[3]（ZCU102 和 ZCU104） | DPUCAHX8L_ISA0（U50、U50LV 和 U280） | DPUCVDX8G_ISA3_C32B3[4](VCK190) | DPUCAHX8H_ISA2_DWC[1]（U50、U55C、U50LV 和 U280） | DPUCADF8H_ISA0（U200 和 U250） | DPUCVDX8H_ISA1_F2W4_4PE[2] (VCK5000) | DPUCV2DX8G_ISA1_C20B1 [5](VEK280/V70) |
|---|---|---|---|---|---|---|---|---|
| 内部参数 | | **channel_parallel：16**<br>bank_depth：2048<br>bank_num：8 | **channel_parallel：32**<br>bank_depth：4096 | **channel_parallel：16**<br>bank_depth：8192<br>bank_num：8 | **channel_parallel：16**<br>bank_depth：2048 | **channel_parallel：16**<br>bank_depth：8192 | **channel_parallel：64**<br>bank_depth：2048 | **channel_parallel：32**<br>bank_depth：65528<br>bank_num：1 |
| depthwise-transposed-conv2d | Kernel size<br>Strides | kernel_w/stride_w, kernel_h/stride_h: [1, 256] | kernel_w/stride_w, kernel_h/stride_h: [3] | kernel_w/stride_w, kernel_h/stride_h: [1, 256] | kernel_w/stride_w, kernel_h/stride_h: {1,2, 3, 5, 7} | 不支持 | kernel_w/stride_w, kernel_h/stride_h: [1, 8] | kernel_w/stride_w, kernel_h/stride_h: [1, 256] |
| | Paddings | pad_left, pad_right: [0, min((kernel_w-1), 15)] | pad_left, pad_right: [1, kernel_w-1] | pad_left, pad_right: [0, min((kernel_w-1),15)] | pad_left, pad_right: [1, kernel_w-1] | | pad_left, pad_right: [1, kernel_w-1] | pad_left, pad_right: [0, min((kernel_w-1),15)] |
| | | pad_top, pad_bottom: [0, min((kernel_h-1), 15)] | pad_top, pad_bottom: [1, kernel_h-1] | pad_top, pad_bottom: [0, min((kernel_h-1), 15)] | pad_top, pad_bottom: [1, kernel_h-1] | | pad_top, pad_bottom: [1, kernel_h-1] | pad_top, pad_bottom: [0, min((kernel_h-1), 15)] |
| | Out Size | output_channel <= 256 * channel_parallel | | | | | output_channel <= 256 * channel_parallel | |
| | Activation | ReLU、ReLU6、LeakyReLU[6]、Hard-Swish 和 Hard-Sigmoid | ReLU 和 ReLU6 | ReLU、ReLU6、LeakyReLU[7]、Hard-Swish 和 Hard-Sigmoid | ReLU 和 ReLU6 | | ReLU 和 ReLU6 | ReLU、ReLU6、LeakyReLU、Hard-Swish 和 Hard-Sigmoid |

表 28：当前支持的运算符 (续)

| CNN 中的典型运算类型 | 参数 | DPUCZDX8G_ISA1_B4096[3]（ZCU102 和 ZCU104） | DPUCAHX8L_ISA0（U50、U50LV 和 U280） | DPUCVDX8G_ISA3_C32B3[4](VCK190) | DPUCAHX8H_ISA2_DWC[1]（U50、U55C、U50LV 和 U280） | DPUCADF8H_ISA0（U200 和 U250） | DPUCVDX8H_ISA1_F2W4_4PE[2](VCK5000) | DPUCV2DX8G_ISA1_C20B1 [5](VEK280/V70) |
|---|---|---|---|---|---|---|---|---|
| 内部参数 | | channel_parallel：16<br>bank_depth：2048<br>bank_num：8 | channel_parallel：32<br>bank_depth：4096 | channel_parallel：16<br>bank_depth：8192<br>bank_num：8 | channel_parallel：16<br>bank_depth：2048 | channel_parallel：16<br>bank_depth：8192 | channel_parallel：64<br>bank_depth：2048 | channel_parallel：32<br>bank_depth：65528<br>bank_num：1 |
| max-pooling | Kernel size | w, h: [1, 256]<br>ceil(h/bank_num) * w <= bank_depth | w, h: {2, 3, 5, 7, 8} | w, h: [1, 256]<br>ceil(h/bank_num) * w <= bank_depth | w, h: [1, 8] | w, h: [1, 16] | w, h: [1, 128] | w, h: [1, 256]<br>h * w <= bank_depth |
| | Strides | w, h: [1, 256] | w, h: [1, 8] | w, h: [1, 256] | w, h: [1, 8] | w, h: [1, 8] | w, h: [1, 128] | w, h: [1, 256] |
| | Paddings | pad_left, pad_right: [0, min((kernel_w-1), 15)] | pad_left, pad_right: [1, kernel_w-1] | pad_left, pad_right: [0, min((kernel_w-1), 15)] | pad_left, pad_right: [1, kernel_w-1] | | | pad_left, pad_right: [0, min((kernel_w-1), 15)] |
| | | pad_top, pad_bottom: [0, min((kernel_h-1), 15)] | pad_top, pad_bottom: [1, kernel_h-1] | pad_top, pad_bottom: [0, min((kernel_h-1), 15)] | pad_top, pad_bottom: [1, kernel_h-1] | | | pad_top, pad_bottom: [0, min((kernel_h-1), 15)] |
| | Activation | ReLU 和 ReLU6 | 不支持 | ReLU 和 ReLU6 | 不支持 | ReLU | 不支持 | ReLU 和 ReLU6 |
| average-pooling | Kernel size | w, h: [1, 256]<br>ceil(h/bank_num) * w <= bank_depth | w, h: {2, 3, 5, 7, 8}<br>w==h | w, h: [1, 256]<br>ceil(h/bank_num) * w <= bank_depth | w, h: [1, 8]<br>w==h | w, h: [1, 16] | w, h: [1, 128]<br>w==h | w, h: [1, 256]<br>h * w <= bank_depth |
| | Strides | w, h: [1, 256] | w, h: [1, 8] | w, h: [1, 256] | w, h: [1, 8] | w, h: [1, 8] | w, h: [1, 128] | w, h: [1, 256] |
| | Paddings | pad_left, pad_right: [0, min((kernel_w-1), 15)] | pad_left, pad_right: [1, kernel_w-1] | pad_left, pad_right: [0, min((kernel_w-1), 15)] | pad_left, pad_right: [1, kernel_w-1] | | | pad_left, pad_right: [0, min((kernel_w-1), 15)] |
| | | pad_top, pad_bottom: [0, min((kernel_h-1), 15)] | pad_top, pad_bottom: [1, kernel_h-1] | pad_top, pad_bottom: [0, min((kernel_h-1), 15)] | pad_top, pad_bottom: [1, kernel_h-1] | | | pad_top, pad_bottom: [0, min((kernel_h-1), 15)] |
| | Activation | ReLU 和 ReLU6 | 不支持 | ReLU 和 ReLU6 | 不支持 | ReLU | 不支持 | ReLU 和 ReLU6 |

表 28：当前支持的运算符 (续)

| CNN 中的典型运算类型 | 参数 | DPUCZDX8G_ISA1_B4096[3]（ZCU102 和 ZCU104） | DPUCAHX8L_ISA0（U50、U50LV 和 U280） | DPUCVDX8G_ISA3_C32B3[4](VCK190) | DPUCAHX8H_ISA2_DWC[1]（U50、U55C、U50LV 和 U280） | DPUCADF8H_ISA0（U200 和 U250） | DPUCVDX8H_ISA1_F2W4_4PE[2] (VCK5000) | DPUCV2DX8G_ISA1_C20B1 [5](VEK280/V70) |
|---|---|---|---|---|---|---|---|---|
| 内部参数 | | **channel_parallel：16**<br>bank_depth：2048<br>bank_num：8 | **channel_parallel：32**<br>bank_depth：4096 | **channel_parallel：16**<br>bank_depth：8192<br>bank_num：8 | **channel_parallel：16**<br>bank_depth：2048 | **channel_parallel：16**<br>bank_depth：8192 | **channel_parallel：64**<br>bank_depth：2048 | **channel_parallel：32**<br>bank_depth：65528<br>bank_num：1 |
| eltwise | type | sum, prod | sum | sum, prod | sum | sum | sum, prod | 2-input sum, prod |
| | Input Channel | input_channel <= 256 * channel_parallel | | | | | | |
| | Activation | ReLU | ReLU | ReLU | ReLU | ReLU | ReLU 和 Hard-Sigmoid | ReLU |
| concat | 特定于网络的限制，与特征图的大小、量化结果和编译器优化有关。 | | | | | | | |
| reorg | Strides | reverse==false : stride ^ 2 * input_channel <= 256 * channel_parallel<br>reverse==true : input_channel <= 256 * channel_parallel | | | | | | |
| pad | In Size | input_channel <= 256 * channel_parallel | | | | | | |
| | Mode | "SYMMETRIC"（在编译器最优化过程中，"CONSTANT" pad(value=0) 会被融合到相邻运算符中） | | | | | "SYMMETRIC"，"CONSTANT"（所有填充值都相同） | "SYMMETRIC"（在编译器最优化过程中，"CONSTANT" pad(value=0) 会被融合到相邻运算符中） |
| global pooling | 全局池化将作为常规池化来处理，内核大小等于输入张量大小。 | | | | | | | |
| InnerProduct、Fully Connected 和 Matmul | 这些运算将会变换为 conv2d 运算 | | | | | | | |

表 28：当前支持的运算符 (续)

| CNN 中的典型运算类型 | 参数 | DPUCZDX8G_ISA1_B4096[3]（ZCU102 和 ZCU104） | DPUCAHX8L_ISA0（U50、U50LV 和 U280） | DPUCVDX8G_ISA3_C32B3[4](VCK190) | DPUCAHX8H_ISA2_DWC[1]（U50、U55C、U50LV 和 U280） | DPUCADF8H_ISA0（U200 和 U250） | DPUCVDX8H_ISA1_F2W4_4PE[2] (VCK5000) | DPUCV2DX8G_ISA1_C20B1 [5](VEK280/V70) |
|---|---|---|---|---|---|---|---|---|
| 内部参数 | | **channel_parallel：16**<br>bank_depth：2048<br>bank_num：8 | **channel_parallel：32**<br>bank_depth：4096 | **channel_parallel：16**<br>bank_depth：8192<br>bank_num：8 | **channel_parallel：16**<br>bank_depth：2048 | **channel_parallel：16**<br>bank_depth：8192 | **channel_parallel：64**<br>bank_depth：2048 | **channel_parallel：32**<br>bank_depth：65528<br>bank_num：1 |
| resize | scale | NEAREST：ceil(scale/bank_num) * scale * ceil(input_channel/channel_parallel) <= bank_depth<br>BILINEAR：仅适用于 4-D 特征映射。这将变换为 pad 和 depthwise-transposed-conv2d。<br>TRILINEAR：仅适用于 5-D 特征映射。这将变换为 pad 和 transposed-conv3d。 | | | | | | |
| | mode | NEAREST 和 BILINEAR | NEAREST 和 BILINEAR | NEAREST、BILINEAR 和 TRILINEAR | NEAREST 和 BILINEAR | NEAREST 和 BILINEAR | NEAREST 和 BILINEAR | NEAREST 和 BILINEAR |

表 28：当前支持的运算符 (续)

| CNN 中的典型运算类型 | 参数 | DPUCZDX8G_ISA1_B4096[3]（ZCU102 和 ZCU104） | DPUCAHX8L_ISA0（U50、U50LV 和 U280） | DPUCVDX8G_ISA3_C32B3[4](VCK190) | DPUCAHX8H_ISA2_DWC[1]（U50、U55C、U50LV 和 U280） | DPUCADF8H_ISA0（U200 和 U250） | DPUCVDX8H_ISA1_F2W4_4PE[2] (VCK5000) | DPUCV2DX8G_ISA1_C20B1 [5](VEK280/V70) |
|---|---|---|---|---|---|---|---|---|
| 内部参数 | | **channel_parallel：16** bank_depth：2048 bank_num：8 | **channel_parallel：32** bank_depth：4096 | **channel_parallel：16** bank_depth：8192 bank_num：8 | **channel_parallel：16** bank_depth：2048 | **channel_parallel：16** bank_depth：8192 | **channel_parallel：64** bank_depth：2048 | **channel_parallel：32** bank_depth：65528 bank_num：1 |
| conv3d | kernel size | 不支持 | 不支持 | w, h, d: [1, 16] w * h * ceil(ceil(input_channel/16) * 16 * d / 2048) <= 64 | 不支持 | 不支持 | 不支持 | 不支持 |
| | strides | | | w, h, d: [1, 8] | | | | |
| | paddings | | | pad_left, pad_right: [0, kernel_w-1] pad_top, pad_bottom: [0, kernel_h-1] pad_front, pad_back: [0, kernel_d-1] | | | | |
| | In size | | | kernel_w * kernel_h * kernel_d * ceil(input_channel/channel_parallel) <= bank_depth, input_channel <= 256 * channel_parallel | | | | |
| | Out size | | | output_channel <= 256 * channel_parallel | | | | |
| | Activation | | | ReLU、LeakyReLU、ReLU6、Hard-Swish 和 Hard-Sigmoid | | | | |

表 28：当前支持的运算符 (续)

| CNN 中的典型运算类型 | 参数 | DPUCZDX8G_ISA1_B4096[3]（ZCU102 和 ZCU104） | DPUCAHX8L_ISA0（U50、U50LV 和 U280） | DPUCVDX8G_ISA3_C32B3[4](VCK190) | DPUCAHX8H_ISA2_DWC[1]（U50、U55C、U50LV 和 U280） | DPUCADF8H_ISA0（U200 和 U250） | DPUCVDX8H_ISA1_F2W4_4PE[2](VCK5000) | DPUCV2DX8G_ISA1_C20B1[5](VEK280/V70) |
|---|---|---|---|---|---|---|---|---|
| 内部参数 | | **channel_parallel：16**<br>bank_depth：2048<br>bank_num：8 | **channel_parallel：32**<br>bank_depth：4096 | **channel_parallel：16**<br>bank_depth：8192<br>bank_num：8 | **channel_parallel：16**<br>bank_depth：2048 | **channel_parallel：16**<br>bank_depth：8192 | **channel_parallel：64**<br>bank_depth：2048 | **channel_parallel：32**<br>bank_depth：65528<br>bank_num：1 |
| depthwise-conv3d | kernel size | 不支持 | 不支持 | w, h: [1, 256]<br>d: [1, 16] | 不支持 | 不支持 | 不支持 | 不支持 |
| | strides | | | w, h: [1, 256]<br>d=1 | | | | |
| | paddings | | | pad_left, pad_right: [0, min((kernel_w-1), 15)]<br>pad_top, pad_bottom: [0, min((kernel_h-1), 15)]<br>pad_front, pad_back: [0, min((kernel_d-1), 15)] | | | | |
| | In size | | | kernel_w * kernel_h * kernel_d * ceil(input_channel/channel_parallel) <= bank_depth | | | | |
| | Out size | | | output_channel <= 256 * channel_parallel | | | | |
| | Activation | | | ReLU 和 ReLU6 | | | | |

表 28：当前支持的运算符 (续)

| CNN 中的典型运算类型 | 参数 | DPUCZDX8G_ISA1_B4096[3]（ZCU102 和 ZCU104） | DPUCAHX8L_ISA0（U50、U50LV 和 U280） | DPUCVDX8G_ISA3_C32B3[4](VCK190) | DPUCAHX8H_ISA2_DWC[1]（U50、U55C、U50LV 和 U280） | DPUCADF8H_ISA0（U200 和 U250） | DPUCVDX8H_ISA1_F2W4_4PE[2] (VCK5000) | DPUCV2DX8G_ISA1_C20B1 [5](VEK280/V70) |
|---|---|---|---|---|---|---|---|---|
| 内部参数 | | **channel_parallel：16**<br>bank_depth：2048<br>bank_num：8 | **channel_parallel：32**<br>bank_depth：4096 | **channel_parallel：16**<br>bank_depth：8192<br>bank_num：8 | **channel_parallel：16**<br>bank_depth：2048 | **channel_parallel：16**<br>bank_depth：8192 | **channel_parallel：64**<br>bank_depth：2048 | **channel_parallel：32**<br>bank_depth：65528<br>bank_num：1 |
| transposed-conv3d | kernel size | 不支持 | 不支持 | kernel_w/stride_w, kernel_h/stride_h, kernel_d/stride_d: [1, 16] | 不支持 | 不支持 | 不支持 | 不支持 |
| | strides | | | | | | | |
| | paddings | | | pad_left, pad_right: [0, kernel_w-1]<br>pad_top, pad_bottom: [0, kernel_h-1]<br>pad_front, pad_back: [0, kernel_d-1] | | | | |
| | Out size | | | output_channel <= 256 * channel_parallel | | | | |
| | Activation | | | ReLU、LeakyReLU、ReLU6、Hard-Swish 和 Hard-Sigmoid | | | | |

表 28：当前支持的运算符 (续)

| CNN 中的典型运算类型 | 参数 | DPUCZDX8G_ISA1_B4096[3]（ZCU102 和 ZCU104） | DPUCAHX8L_ISA0（U50、U50LV 和 U280） | DPUCVDX8G_ISA3_C32B3[4](VCK190) | DPUCAHX8H_ISA2_DWC[1]（U50、U55C、U50LV 和 U280） | DPUCADF8H_ISA0（U200 和 U250） | DPUCVDX8H_ISA1_F2W4_4PE[2] (VCK5000) | DPUCV2DX8G_ISA1_C20B1 [5](VEK280/V70) |
|---|---|---|---|---|---|---|---|---|
| 内部参数 | | **channel_parallel：16**<br>bank_depth：2048<br>bank_num：8 | **channel_parallel：32**<br>bank_depth：4096 | **channel_parallel：16**<br>bank_depth：8192<br>bank_num：8 | **channel_parallel：16**<br>bank_depth：2048 | **channel_parallel：16**<br>bank_depth：8192 | **channel_parallel：64**<br>bank_depth：2048 | **channel_parallel：32**<br>bank_depth：65528<br>bank_num：1 |
| depthwise-transposed-conv3d | kernel size<br>strides | 不支持 | 不支持 | kernel_w/stride_w, kernel_h/stride_h, kernel_d/stride_d: [1, 16] | 不支持 | 不支持 | 不支持 | 不支持 |
| | paddings | | | pad_left, pad_right: [0, min((kernel_w-1), 15)]<br>pad_top, pad_bottom: [0, min((kernel_h-1), 15)]<br>pad_front, pad_back: [0, min((kernel_d-1), 15)] | | | | |
| | Out size | | | output_channel <= 256 * channel_parallel | | | | |
| | Activation | | | ReLU 和 ReLU6 | | | | |
| Strided_slice | Stride | Stride_batch = 1<br>Stride_channel = 1 | | | | | | |
| correlation1d_elemwise | input size | input_channel <= 256 * channel_parrallel | 不支持 | input_channel <= 256 * channel_parrallel | 不支持 | 不支持 | 不支持 | 不支持 |
| correlation2d_elemwise | input size | input_channel <= 256 * channel_parrallel | 不支持 | input_channel <= 256 * channel_parrallel | 不支持 | 不支持 | 不支持 | 不支持 |

表 28：当前支持的运算符 (续)

| CNN 中的典型运算类型 | 参数 | DPUCZDX8G_ISA1_B4096[3]（ZCU102 和 ZCU104） | DPUCAHX8L_ISA0（U50、U50LV 和 U280） | DPUCVDX8G_ISA3_C32B3[4](VCK190) | DPUCAHX8H_ISA2_DWC[1]（U50、U55C、U50LV 和 U280） | DPUCADF8H_ISA0（U200 和 U250） | DPUCVDX8H_ISA1_F2W4_4PE[2](VCK5000) | DPUCV2DX8G_ISA1_C20B1 [5](VEK280/V70) |
|---|---|---|---|---|---|---|---|---|
| 内部参数 | | **channel_parallel：16**<br>bank_depth：2048<br>bank_num：8 | **channel_parallel：32**<br>bank_depth：4096 | **channel_parallel：16**<br>bank_depth：8192<br>bank_num：8 | **channel_parallel：16**<br>bank_depth：2048 | **channel_parallel：16**<br>bank_depth：8192 | **channel_parallel：64**<br>bank_depth：2048 | **channel_parallel：32**<br>bank_depth：65528<br>bank_num：1 |
| argmax | axis | axis = input_channel | 不支持 | axis = input_channel | 不支持 | 不支持 | 不支持 | axis = input_channel |
| | input size | input_channel < =128 | | input_channel < =128 | | | | input_channel < =128 |
| reduction max | axis | axis = input_channel | 不支持 | axis = input_channel | 不支持 | 不支持 | 不支持 | axis = input_channel |
| | input size | input_channel < 2 ^ 12 | | input_channel < 2 ^ 12 | | | | input_channel < 2 ^ 12 |
| cost_volume | input size | input_channel <= 256 * channel_parallel | 不支持 | input_channel <= 256 * channel_parallel | 不支持 | 不支持 | 不支持 | 不支持 |
| transpose | | | | | | | | |

**注释：**

1. 对于 DPUCAHX8H，此处仅列出 DPUCAHX8H_ISA2_DWC。如需查看更多 IP 配置，请参阅《适用于卷积神经网络的 DPUCAHX8H 产品指南》(PG367)
2. 对于 DPUCVDX8H，此处仅列出 DPUCVDX8H_ISA1_F2W4_4PE。如需查看更多 IP 配置，请参阅《适用于卷积神经网络的 DPUCVDX8H LogiCORE IP》(PG403)
3. 对于 DPUCZDX8G，此处仅列出 DPUCZDX8G_ISA1_B4096。如需查看更多 IP 配置，请参阅《适用于 Zynq UltraScale+ MPSoC 的 DPUCZDX8G》(PG338)
4. 对于 DPUCVDX8G，此处仅列出 DPUCVDX8G_ISA3_C32B3。如需查看更多 IP 配置，请参阅《适用于 Versal 自适应 SoC 的 DPUCVDX8G 产品指南》(PG389)
5. 对于 DPUCV2DX8G，此处仅列出 DPUCV2DX8G_ISA1_C20B1。如需查看更多 IP 配置，请参阅《适用于 Versal 自适应 SoC 的 DPUCV2DX8G 产品指南》(PG425)
6. 对于 DPUCZDX8G，默认对于类 depthwise-conv 运算符，不启用激活 LeakyReLU。如需了解如何启用此激活，请参阅《适用于 Zynq UltraScale+ MPSoC 的 DPUCZDX8G》(PG338)
7. 对于 DPUCVDX8G，默认对于类 depthwise-conv 运算符，不启用激活 LeakyReLU。如需了解如何启用此激活，请参阅《适用于 Versal 自适应 SoC 的 DPUCVDX8G 产品指南》(PG389)

## 受 TensorFlow 支持的运算符

表 29：**受 TensorFlow 支持的运算符**

| TensorFlow | | XIR | | DPU 实现 |
|---|---|---|---|---|
| **OP 类型** | **属性** | **OP 名称** | **属性** | |
| placeholder / inputlayer* | shape | data | shape | 为输入数据分配存储器。 |
| | | | data_type | |
| const | | const | data | 为常量数据分配存储器。 |
| | | | shape | |
| | | | data_type | |
| conv2d | filter | conv2d | kernel | 卷积引擎。 |
| | strides | | stride | |
| | | | pad([0, 0, 0, 0]) | |
| | padding | | pad_mode（SAME 或 VALID） | |
| | dilations | | dilation | |
| conv2d* | kernel_size | conv2d | kernel | |
| | strides | | stride | |
| | padding | | pad([0, 0, 0, 0]) | |
| | dilation_rate | | dilation | |
| | use_bias | | | |
| | group | | group | |
| depthwiseconv2dnative | filter | depthwise-conv2d | kernel | 逐通道卷积引擎。 |
| | strides | | stride | |
| | explicit_paddings | | pad | |
| | padding | | pad_mode（SAME 或 VALID） | |
| | dilations | | dilation | |
| conv2dbackpropinput / conv2dtranspose* | filter | transposed-conv2d | kernel | 卷积引擎。 |
| | strides | | stride | |
| | | | pad([0, 0, 0, 0]) | |
| | padding | | pad_mode（SAME 或 VALID） | |
| | dilations | | dilation | |
| spacetobacthnd + conv2d + batchtospacend | block_shape | conv2d | dilation | 满足 AMD 设定的特定要求时，Spacetobatch、Conv2d 和 Batchtospace 将被映射到卷积引擎。 |
| | padding | | pad | |
| | filter | | kernel | |
| | strides | | stride | |
| | padding | | pad_mode(SAME) | |
| | dilations | | dilation | |
| | block_shape | | | |
| | crops | | | |

表 29：受 **TensorFlow** 支持的运算符 (续)

| TensorFlow | | XIR | | DPU 实现 |
|---|---|---|---|---|
| OP 类型 | 属性 | OP 名称 | 属性 | |
| matmul / dense* | transpose_a | conv2d / matmul | transpose_a | 当等效 conv2d 满足硬件要求并且可映射到 DPU 后，matmul 将立即转换为 conv2d 运算。 |
| | transpose_b | | transpose_b | |
| maxpool / maxpooling2d* / globalmaxpool2d* | ksize | maxpool2d | kernel | 池化引擎。当原始池化运算符需全局减法时，global 属性将设为 true。 |
| | strides | | stride | |
| | | | pad([0, 0, 0, 0]) | |
| | padding | | pad_mode（SAME 或 VALID） | |
| | | | global | |
| avgpool / averagepooling2d* / globalavgeragepooling2d* | pool_size | avgpool2d | kernel | 池化引擎。当原始池化运算符需全局减法时，global 属性将设为 true。 |
| | strides | | stride | |
| | | | pad([0, 0, 0, 0]) | |
| | padding | | pad_mode（SAME 或 VALID） | |
| | | | count_include_pad (false) | |
| | | | count_include_invalid (true) | |
| | | | global | |
| mean | axis | avgpool / reduction_mean | axis | 如果等效 avgpool 满足硬件要求并且可映射到 DPU，mean 运算将被转换为 avgpool。 |
| | keep_dims | | keep_dims | |
| relu | | relu | | 激活将被融合到相邻运算（如 convolution）。 |
| relu6 | | relu6 | | |
| leakyrelu | alpha | leaky_relu | alpha | |
| fixneuron / quantizelayer* | bit_width | fix | bit_width | 编译期间，它将被分为 float2fix 和 fix2float，然后 float2fix 和 fix2float 运算将与相邻运算融合为低精度运算。 |
| | quantize_pos | | fix_point | |
| | | | if_signed | |
| | | | round_mode | |
| identity | | identity | | Identity 将被移除。 |
| add、addv2 | | add | | 如果 add 为逐元素加法，那么 add 将被映射到 DPU 逐元素加法引擎；如果 add 为逐通道加法，AMD 会伺机将 add 与相邻运算（例如，convolution）融合。 |

表 29：受 **TensorFlow** 支持的运算符 (续)

| TensorFlow | | XIR | | DPU 实现 |
|---|---|---|---|---|
| **OP 类型** | **属性** | **OP 名称** | **属性** | |
| mul | | mul | | 只要 Mul 所含任一输入为常量，即可映射到 Depthwise-Convolution Engine（逐通道卷积引擎）。如果其两个输入为相同形状，则可将其作为逐元素乘法映射到 Misc Engine（其他引擎）。如果另有某些 mul 运算属于特殊运算符组合的一部分，则此 mul 可融合到这些组合内。否则，它将被映射到 CPU。 |
| concatv2 / concatenate* | axis | concat | axis | AMD 通过特殊的读取或写入策略和谨慎分配片上存储器来减少源于 concat 的开销。 |
| pad / zeropadding2d* | paddings | pad | paddings | 首个编译器将尝试把“CONSTANT”填充融合到相邻运算中，例如，卷积和池化。如不存在此类运算符，那么当填充维度等于 4 且满足硬件要求时，仍可将其映射到 DPU。对于“SYMMETRIC”填充，它将映射到 DPU。但 DPU 不支持“REFLECT”填充。 |
| | mode | | mode | |
| | | | constant_values | |
| shape | | shape | | shape 运算将被移除。 |
| stridedslice | begin | strided_slice | begin | 如果这些运算与 shape 相关，那么在编译期间将被移除。如果属于低精度运算，则将与相邻运算融合。否则，这些运算将被编译到 CPU 实现中。 |
| | end | | end | |
| | strides | | strides | |
| pack | axis | stack | axis | |
| neg | | neg | | |
| realdiv | | div | | |
| sub | | sub | | |
| prod | axis | reduction_product | axis | |
| | keep_dims | | keep_dims | |
| sum | axis | reduction_sum | axis | |
| | keep_dims | | keep_dims | |
| max | axis | reduction_max | axis | |
| | keep_dims | | keep_dims | |

表 29：**受 TensorFlow 支持的运算符** (续)

| TensorFlow | | XIR | | DPU 实现 |
|---|---|---|---|---|
| **OP 类型** | **属性** | **OP 名称** | **属性** | |
| resizebilinear | size | resize | size | 如果 resize 的模式为“BILINEAR”，那么 align_corner=false, half_pixel_centers = false, size = 2, 4, 8; align_corner=false, half_pixel_centers = true, size = 2, 4 可转换为 DPU 实现 (pad +depthwise-transposed conv2d)。如果 resize 模式为“NEAREST”，那么 size 为整数，且 resize 将被映射到 DPU 实现。 |
| | align_corners | | align_corners | |
| | half_pixel_centers | | half_pixel_centers | |
| | | | mode="BILINEAR" | |
| resizenearestneighbor | size | resize | size | |
| | align_corners | | align_corners | |
| | half_pixel_centers | | half_pixel_centers | |
| | | | mode="NEAREST" | |
| upsample2d/ upsampling2d* | size | resize | scale | |
| | | | align_corners | |
| | | | half_pixel_centers | |
| | interpolation | | mode | |
| reshape | shape | reshape | shape | 在某些情况下，将被转换为 reshape 运算。否则，将被映射到 CPU。 |
| reshape* | target_shape | | | |
| transpose | perm | transpose | order | |
| squeeze | axis | squeeze | axis | |
| exp | | exp | | 将仅被编译到 CPU 实现中。 |
| softmax | axis | softmax | axis | |
| sigmoid | | sigmoid | | |
| square+ rsqrt+ maximum | | l2_normalize | axis | output = x / sqrt(max(sum(x ^ 2), epsilon)) 将融合到 XIR 中的 l2_normalize。 |
| | | | epsilon | |

**注释：**

1. TensorFlow 中以上列出的运算 (OP) 在 XIR 中均受支持。所有这些运算在工具链中都有 CPU 实现。
2. 含 * 的运算符表示 TensorFlow 版本高于 2.0。

# 受 PyTorch 支持的运算符

表 30：**受 PyTorch 支持的运算符**

| PyTorch | | XIR | | DPU 实现 |
|---|---|---|---|---|
| **API** | **属性** | **OP 名称** | **属性** | |
| 参数/张量/零 | data | const | data | 为输入数据分配存储器。 |
| | | | shape | |
| | | | data_type | |

表 30：受 **PyTorch** 支持的运算符 (续)

| PyTorch | | XIR | | DPU 实现 |
|---|---|---|---|---|
| **API** | **属性** | **OP 名称** | **属性** | |
| Conv2d | in_channels | conv2d (groups = 1) / depthwise-conv2d (groups = input channel) | | 如果 groups == input channel，那么卷积将被编译到逐通道卷积引擎中。如果 groups == 1，那么卷积将映射到卷积引擎。否则，它将被映射到 CPU。 |
| | out_channels | | | |
| | kernel_size | | kernel | |
| | stride | | stride | |
| | padding | | pad | |
| | padding_mode('zeros') | | pad_mode (FLOOR) | |
| | groups | | | |
| | dilation | | dilation | |
| ConvTranspose2d | in_channels | transposed-conv2d (groups = 1) / depthwise-transposed-conv2d (groups = input channel) | | 如果 groups == input channel，那么卷积将被编译到逐通道卷积引擎中。如果 groups == 1，那么卷积将映射到卷积引擎。否则，它将被映射到 CPU。对于 output_padding 特征，尚未支持 DPU，因此如果值并非全部为 0，则该运算符将分配给 CPU。 |
| | out_channels | | | |
| | kernel_size | | kernel | |
| | stride | | stride | |
| | padding | | pad | |
| | output_padding | | output_padding | |
| | padding_mode('zeros') | | pad_mode (FLOOR) | |
| | groups | | | |
| | dilation | | dilation | |
| matmul | | conv2d / matmul | transpose_a | matmul 将被转换为 conv2d 并编译为卷积引擎。如果 matmul 转换失败，则将由 CPU 实现。 |
| | | | transpose_b | |
| MaxPool2d / AdaptiveMaxPool2d | kernel_size | maxpool2d | kernel | 池化引擎 |
| | stride | | stride | |
| | padding | | pad | |
| | ceil_mode | | pad_mode | |
| | output_size (adaptive) | | global | |
| AvgPool2d / AdaptiveAvgPool2d | kernel_size | avgpool2d | kernel | 池化引擎 |
| | stride | | stride | |
| | padding | | pad | |
| | ceil_mode | | pad_mode | |
| | count_include_pad | | count_include_pad | |
| | | | count_include_invalid (true) | |
| | output_size (adaptive) | | global | |

表 30：受 **PyTorch** 支持的运算符 (续)

| PyTorch | | XIR | | DPU 实现 |
|---|---|---|---|---|
| API | 属性 | OP 名称 | 属性 | |
| ReLU | | relu | | 激活将被融合到相邻运算（如 convolution）。 |
| LeakyReLU | negative_slope | leakyrelu | alpha | |
| ReLU6 | | relu6 | | |
| Hardtanh | min_val = 0 | | | |
| | max_val = 6 | | | |
| Hardsigmoid | | hard-sigmoid | | |
| Hardswish | | hardswish | | |
| ConstantPad2d / ZeroPad2d | padding | pad | paddings | 首个编译器将尝试把“CONSTANT”填充融合到相邻运算中，例如，卷积和池化。如不存在此类运算符，那么当填充维度等于 4 且满足硬件要求时，仍可将其映射到 DPU。 |
| | value = 0 | | constant_values | |
| | | | mode ("CONSTANT") | |
| add | | add | | 如果 add 为逐元素加法，add 将映射到 DPU 逐元素加法引擎。如果 add 为逐通道加法，则伺机将 add 与相邻运算（例如，convolution）融合。如果这些运算与 shape 相关，那么在编译期间将被移除。如果属于低精度运算，则将与相邻运算融合。否则，这些运算将被编译到 CPU 实现中。只要 Mul 所含任一输入为常量，即可映射到 Depthwise-Convolution Engine（逐通道卷积引擎）。如果其两个输入为相同形状，则可将其作为逐元素乘法映射到 Misc Engine（其他引擎）。如果另有某些 mul 运算属于特殊运算符组合的一部分，则此 mul 可融合到这些组合内。否则，它将被映射到 CPU。 |
| sub / rsub | | sub | | |
| mul | | mul | | |
| neg | | neg | | |
| sum | dim | reduction_sum | axis | |
| | keepdim | | keep_dims | |
| max | dim | reduction_max | axis | |
| | keepdim | | keep_dims | |
| mean | dim | reduction_mean | axis | |
| | keepdim | | keep_dims | |

表 30：受 **PyTorch** 支持的运算符 (续)

| PyTorch | | XIR | | DPU 实现 |
|---|---|---|---|---|
| **API** | **属性** | **OP 名称** | **属性** | |
| interpolate / upsample / upsample_bilinear / upsample_nearest | size | resize | size | 如果 resize 的模式为“BILINEAR”，那么 align_corner=false, half_pixel_centers = false, size = 2, 4, 8; align_corner=false, half_pixel_centers = true, size = 2, 4 可转换为 DPU 实现 (pad +depthwise-transposed conv2d)。如果 resize 模式为“NEAREST”，那么 size 为整数，且 resize 将被映射到 DPU 实现。 |
| | scale_factor | | | |
| | mode | | mode | |
| | align_corners | | align_corners | |
| | | | half_pixel_centers = ! align_corners | |
| transpose | dim0 | transpose | order | 在某些情况下，这些运算将被转换为 reshape 运算。此外，还会伺机将维度转换运算融合到相邻运算的特殊加载或保存指令中以减少开销。否则，将被映射到 CPU。 |
| | dim1 | | | |
| permute | dims | | | |
| view/reshape | size | reshape | shape | |
| flatten | start_dim | reshape/flatten | start_axis | |
| | end_dim | | end_axis | |
| squeeze | dim | reshape / squeeze | axis | |
| cat | dim | concat | axis | 通过特殊的读取或写入策略和谨慎分配片上存储器来减少源于 concat 的开销。 |
| aten::slice* | dim | strided_slice | | 如果 strided_slice 与 shape 相关或者属于低精度运算，则将被移除。否则，strided_slice 将被编译到 CPU 实现中。 |
| | start | | begin | |
| | end | | end | |
| | step | | strides | |
| BatchNorm2d | eps | depthwise-conv2d / scale | epsilon | 如果 batch_norm 已量化并且可等效转换为 depthwise-conv2d，那么它将被转换为 depthwise-conv2d 并且编译器会伺机执行编译以将 batch_norm 映射到 DPU 实现。否则，batch_norm 将由 CPU 执行。 |
| | | | axis | |
| | | | moving_mean | |
| | | | moving_var | |
| | | | gamma | |
| | | | beta | |
| softmax | dim | softmax | axis | 将仅被编译到 CPU 实现中。 |
| Tanh | | tanh | | |
| Sigmoid | | sigmoid | | |

表 30：受 **PyTorch** 支持的运算符 (续)

| PyTorch | | XIR | | DPU 实现 |
|---|---|---|---|---|
| **API** | **属性** | **OP 名称** | **属性** | |
| PixelShuffle | upscale_factor | pixel_shuffle | scale | 如果卷积为 input，则会将其变换为 tile。 |
| | | | upscale=True | |
| PixelUnshuffle | downscale_factor | pixel_shuffle | scale | |
| | | | upscale=False | |

**注释：**

1. 如果 PyTorch 中的张量分片是以 Python 语法编写的，那么它将被转换为 `aten::slice`。

# VAI_C 用法

在数据中心到边缘 DPU 上，Caffe 和 TensorFlow 框架的对应 Vitis AI 编译器分别为 `vai_c_caffe`、`vai_c_tensorflow`、`vai_c_tensorflow2` 和 `vai_c_xir`。VAI_C 的标准选项如下表所示。

表 31：**用于数据中心和边缘 DPU 的 VAI_C 常用选项**

| 参数 | 描述 |
|---|---|
| --arch | 适用于 VAI_C 编译器的 DPU 架构配置文件，采用 JSON 格式。对于 AMD Vitis™ AI 版本中的预构建 DPU XCLBIN，您可以在 AMD Vitis™ AI Docker (`/opt/vitis_ai/compiler/arch`) 中找到对应的 `arch.json` 文件。内容应类似这样：{"target": "DPUCZDX8G_ISA0_B4096"}。对于自定义 DPU IP，对应的 `arch.json` 文件由 DPU 参考设计和 DPU IP 一起生成。内容应类似这样：{"fingerprint":"0x0101000016010407"}。fingerprint 为 64 位数字签名，用于识别 DPU 目标。其中，1 字节指示 DPU 类型，1 字节指示 ISA 版本，6 字节指示具体配置。每个 DPU 配置的 fingerprint 都是唯一的，运行时依赖它来识别当前平台上运行的 DPU 实例，并验证模型是否针对该 DPU 目标而编译。“DPUCZDX8G_ISA0_B4096”是特定 fingerprint 的别名，在编译器中预定义。 |
| --output_dir | 编译进程完成后的 vai_c_tensorflow 输出目录路径。 |
| --net_name | 由 VAI_C 编译的网络模型的 DPU 内核名称。 |
| --options | 附加选项列表，格式为：'key':'value'。如果要指定多个选项，请用“,”隔开。<br>使用 --options '{"input_shape": "1,224,224,3"}' 手动指定输入形状。<br>使用 --options '{"plugins": "plugin0,plugin1"}' 指定插件库。<br>使用 --options '{"output_ops": "op_name0,op_name1"}' 指定输出运算。<br>使用 --options '{"prefetch": "true"}' 启用跨层预取。<br>使用 --options '{"hd_opt": "true"}' 为 HD 输入启用特殊最优化。<br>**注释：**以“--options”指定的实参享有最高优先级，且覆盖其他位置指定的值。 |

第 5 章

# 部署和运行模型

## 使用 VART 进行编程

AMD Vitis™ AI 提供了 C++ DpuRunner 类，其中包含以下接口：

1. ```
   std::pair<uint32_t, int> execute_async(
                            const std::vector<TensorBuffer*>& input,
                            const std::vector<TensorBuffer*>& output);
   ```

   提交输入张量以供执行，提交输出张量则用于存储结果。主机指针是使用 TensorBuffer 对象来传递的。该函数会返回作业 ID 以及函数调用的状态。

2. ```
   int wait(int jobid, int timeout);
   ```

   由 `execute_async` 返回的作业 ID 将被传递给 `wait()` 以供阻塞，直至作业完成且结果就绪为止。

3. ```
   TensorFormat get_tensor_format()
   ```

   查询 DpuRunner 可知其期望的张量格式。

   返回 DpuRunner::TensorFormat::NCHW 或 DpuRunner::TensorFormat::NHWC

4. ```
   std::vector<Tensor*> get_input_tensors()
   std::vector<Tensor*> get_output_tensors()
   ```

   查询 DpuRunner 可知其针对加载的 Vitis AI 模型所期望的输入和输出张量的形状和名称。

5. 要创建 DpuRunner 对象，请调用以下函数

   ```
   create_runner(const xir::Subgraph* subgraph, const std::string& mode =
   "")
   ```

   它会返回：

   ```
   std::unique_ptr<Runner>
   ```

   create_runner 的输入为由 Vitis AI 编译器生成的 XIR 子计算图。

**提示：**要随 VART 启用多线程，请为每个线程创建 1 个运行器。

**注释：**如果模型有多个子计算图，请参阅

https://github.com/Xilinx/Vitis-AI-Tutorials/tree/3.0/Tutorials/pytorch-subgraphs/。

## C++ 示例

```
// get dpu subgraph by parsing model file
auto runner = vart::Runner::create_runner(subgraph, "run");
// populate input/output tensors
auto job_data = runner->execute_async(inputs, outputs);
runner->wait(job_data.first, -1);
// process outputs
```

Vitis AI 还提供了 Python ctypes Runner 类，用于生成 C++ 类的镜像，它使用的是 C DpuRunner 实现：

```
class Runner:
def __init__(self, path)
def get_input_tensors(self)
def get_output_tensors(self)
def get_tensor_format(self)
def execute_async(self, inputs, outputs)
# differences from the C++ API:
# 1. inputs and outputs are numpy arrays with C memory layout
#    the numpy arrays should be reused as their internal buffer
#    pointers are passed to the runtime. These buffer pointers
#    may be memory-mapped to the FPGA DDR for performance.
# 2. returns job_id, throws exception on error
def wait(self, job_id)
```

## Python 示例

```
dpu_runner = runner.Runner(subgraph, "run")
# populate input/output tensors
jid = dpu_runner.execute_async(fpgaInput, fpgaOutput)
dpu_runner.wait(jid)
# process fpgaOutput
```

# 使用 VART 进行 DPU 调试

本节演示了如何使用 VART 工具来验证 DPU 推断结果。此处使用 TensorFlow、ResNet50 和 PyTorch ResNet50 网络作为示例。以下是使用 VART 进行 DPU 调试的 4 个步骤：

1. 生成量化推断模型和参考结果。
2. 生成 DPU XMODEL。
3. 生成 DPU 推断结果。
4. 请对参考结果和 DPU 推断结果进行交叉核对。

开始调试 DPU 结果前，请确保已根据 https://xilinx.github.io/Vitis-AI/docs/quickstart/vek280.html 中的指示信息完成环境设置。

**注释：**从 AMD Vitis™ AI 2.5 起，弃用 Caffe。如需了解有关 Caffe 的信息，请参阅《Vitis AI 2.0 用户指南》。

# TensorFlow 工作流程

要生成量化推断模型和参考结果，请遵循下列步骤进行操作：

1. 运行以下命令对模型进行量化，以生成量化推断模型。

   这样即可在 `quantize_model` 文件夹中生成量化模型 `quantize_eval_model.pb`。

```
vai_q_tensorflow quantize                                    \
    --input_frozen_graph ./float/resnet_v1_50_inference.pb   \
    --input_fn input_fn.calib_input                          \
    --output_dir quantize_model                              \
    --input_nodes input                                      \
    --output_nodes resnet_v1_50/predictions/Reshape_1        \
    --input_shapes    ?,224,224,3                            \
    --calib_iter    100
```

2. 运行以下命令生成参考数据，以生成参考结果。

```
vai_q_tensorflow dump --input_frozen_graph         \
            quantize_model/quantize_eval_model.pb \
     --input_fn input_fn.dump_input               \
     --output_dir=dump_gpu
```

   下图显示了部分参考数据。

```
input_aquant.bin
input_aquant.txt
resnet_v1_50_Pad_aquant.bin
resnet_v1_50_Pad_aquant.txt
resnet_v1_50_SpatialSqueeze_aquant.bin
resnet_v1_50_SpatialSqueeze_aquant.txt
resnet_v1_50_block1_unit_1_bottleneck_v1_Relu_aquant.bin
resnet_v1_50_block1_unit_1_bottleneck_v1_Relu_aquant.txt
resnet_v1_50_block1_unit_1_bottleneck_v1_conv1_Relu_aquant.bin
resnet_v1_50_block1_unit_1_bottleneck_v1_conv1_Relu_aquant.txt
resnet_v1_50_block1_unit_1_bottleneck_v1_conv2_Relu_aquant.bin
resnet_v1_50_block1_unit_1_bottleneck_v1_conv2_Relu_aquant.txt
resnet_v1_50_block1_unit_1_bottleneck_v1_conv3_BatchNorm_FusedBatchNorm_add_aquant.bin
resnet_v1_50_block1_unit_1_bottleneck_v1_conv3_BatchNorm_FusedBatchNorm_add_aquant.txt
resnet_v1_50_block1_unit_1_bottleneck_v1_shortcut_BatchNorm_FusedBatchNorm_add_aquant.bin
resnet_v1_50_block1_unit_1_bottleneck_v1_shortcut_BatchNorm_FusedBatchNorm_add_aquant.txt
resnet_v1_50_block1_unit_2_bottleneck_v1_Relu_aquant.bin
resnet_v1_50_block1_unit_2_bottleneck_v1_Relu_aquant.txt
resnet_v1_50_block1_unit_2_bottleneck_v1_conv1_Relu_aquant.bin
resnet_v1_50_block1_unit_2_bottleneck_v1_conv1_Relu_aquant.txt
resnet_v1_50_block1_unit_2_bottleneck_v1_conv2_Relu_aquant.bin
resnet_v1_50_block1_unit_2_bottleneck_v1_conv2_Relu_aquant.txt
resnet_v1_50_block1_unit_2_bottleneck_v1_conv3_BatchNorm_FusedBatchNorm_add_aquant.bin
resnet_v1_50_block1_unit_2_bottleneck_v1_conv3_BatchNorm_FusedBatchNorm_add_aquant.txt
resnet_v1_50_block1_unit_3_bottleneck_v1_Pad_aquant.bin
resnet_v1_50_block1_unit_3_bottleneck_v1_Pad_aquant.txt
resnet_v1_50_block1_unit_3_bottleneck_v1_Relu_aquant.bin
resnet_v1_50_block1_unit_3_bottleneck_v1_Relu_aquant.txt
resnet_v1_50_block1_unit_3_bottleneck_v1_conv1_Relu_aquant.bin
resnet_v1_50_block1_unit_3_bottleneck_v1_conv1_Relu_aquant.txt
resnet_v1_50_block1_unit_3_bottleneck_v1_conv2_Relu_aquant.bin
resnet_v1_50_block1_unit_3_bottleneck_v1_conv2_Relu_aquant.txt
```

3. 运行以下命令生成 DPU xmodel 文件，以生成 DPU xmodel，例如 V70。

```
vai_c_tensorflow --frozen_pb quantize_model/quantize_eval_model.pb \
  --arch /opt/vitis_ai/compiler/arch/DPUCV2DX8G/V70/arch.json       \
  --output_dir compile_model                                       \
  --net_name resnet50_tf
```

4. 运行以下命令以生成 DPU 推断结果，并将 DPU 推断结果与参考数据自动匹配。

```
env XLNX_ENABLE_DUMP=1 XLNX_ENABLE_DEBUG_MODE=1 XLNX_GOLDEN_DIR=./
dump_gpu/dump_results_0 \
   xdputil run ./compile_model/resnet_v1_50_tf.xmodel          \
   ./dump_gpu/dump_results_0/input_aquant.bin                  \
   2>result.log 1>&2
```

   如需了解有关 `xdputil` 用法的更多信息，请执行 `xdputil --help` 命令。

   运行以上命令后，即可生成 DPU 推断结果和比较结果 `result.log`。DPU 推断结果位于 `dump` 文件夹中。

5. 请对参考结果和 DPU 推断结果进行交叉核对。

   a. 查看所有层次的比较结果。

```
grep --color=always 'XLNX_GOLDEN_DIR.*layer_name' result.log
```

   b. 仅查看失败的层次。

```
grep --color=always 'XLNX_GOLDEN_DIR.*fail ! layer_name' result.log
```

   如果交叉核对失败，请使用以下方法进一步检查交叉核对失败的层次。

   a. 检查 DPU 和 GPU 的输入。确保它们使用相同的输入数据。

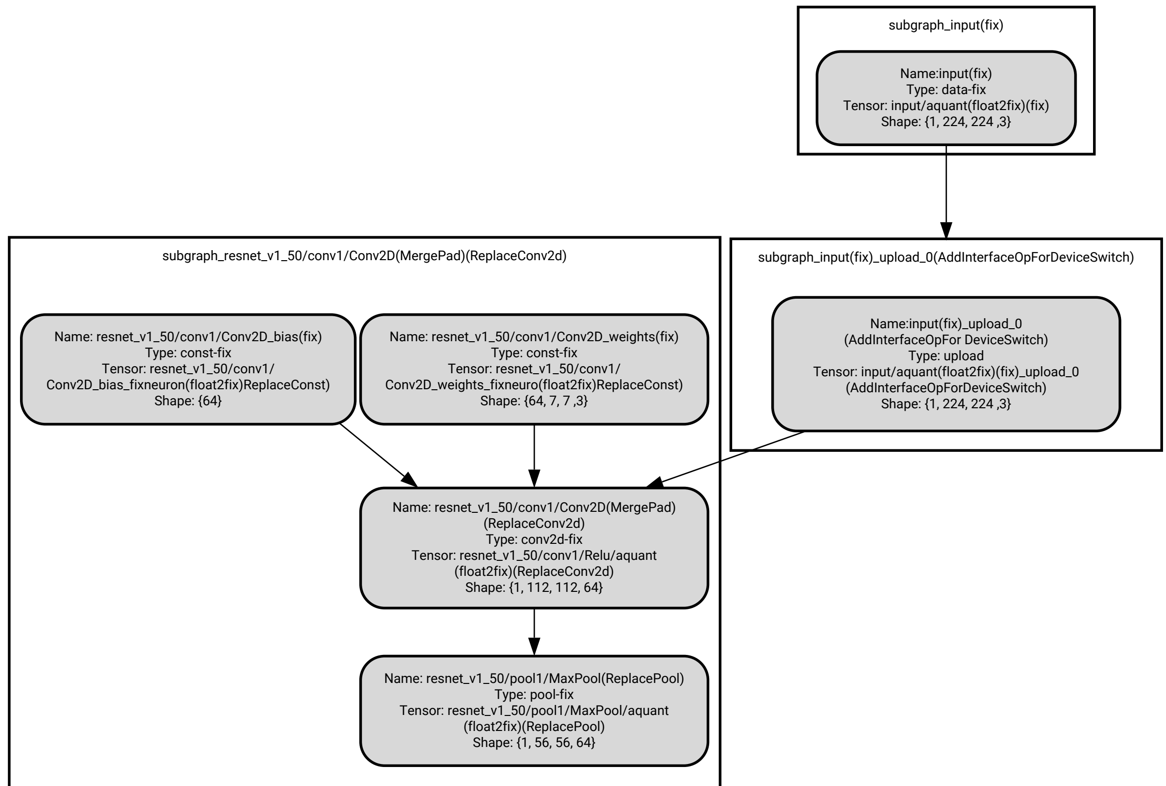
   b. 使用 `xdputil` 工具生成一幅显示网络结构的图像。

```
Usage: xdputil xmodel <xmodel> -s <svg>
```

   **注释：**在 Vitis AI Docker 环境中，执行以下命令即可安装所需的库。

```
sudo apt-get install graphviz
```

   打开您创建的图像后，即可看到这些运算周围的诸多小框。每个框代表 DPU 上的 1 个层次。您可使用最后一项运算的名称在 GPU 转储结果中查找其对应的层次。下图显示了部分架构。




c. 将文件提交给 AMD。

   如果在 DPU 上证明某个特定层存在错误，请准备好量化模型（例如 `quantize_eval_model.pb`），将其打包并与详细描述一起发送给 AMD，以供工厂进行进一步分析。

## PyTorch 工作流程

要生成量化推断模型和参考结果，请遵循下列步骤进行操作：

1. 运行以下命令对模型进行量化，以生成量化推断模型。

```
python resnet18_quant.py --quant_mode calib --subset_len 200
```

2. 在 `export_xmodel` API 中，将 `deploy_check` 设置为 `True`。

```
quantizer.export_xmodel(deploy_check=True)
```

3. 运行以下命令生成参考数据，以生成参考结果。

```
python resnet18_quant.py --quant_mode test  --deploy
```

4. 运行以下命令生成 DPU XMODEL 文件，以生成 DPU XMODEL 文件。

```
vai_c_xir -x /PATH/TO/quantized.xmodel -a /PATH/TO/
arch.json -o /OUTPUTPATH -n netname}
```

5. 生成 DPU 推断结果。

   此步骤与 TensorFlow 工作流程中的步骤相同。

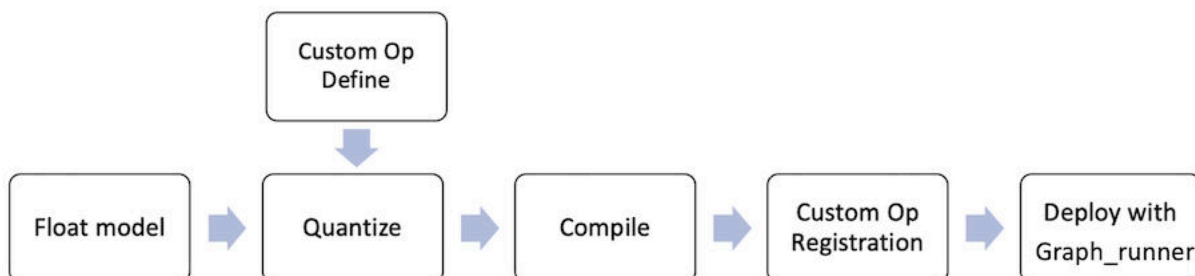
6. 请对参考结果和 DPU 推断结果进行交叉核对。

   此步骤与 TensorFlow 工作流程中的步骤相同。

# 自定义运算符工作流程

从 Vitis AI v2.5 起，支持含自定义运算符 (OP) 的 Pytorch 和 Tensorflow2 模型。自定义运算符的基本工作流程如下所示。

图 23：**自定义运算符工作流程**




此工作流程中的步骤如下所示：

1. 将运算符定义为对 XIR 未知的自定义运算符，然后量化模型。
2. 编译量化的模型。
3. 寄存并实现自定义运算符。
4. 使用 `graph_runner` API 部署模型。

步骤 3 支持 `C++` 和 `Python` 实现并寄存自定义运算符。Vitis AI Library 支持 50 余种常用 OP。您可在 https://github.com/Xilinx/Vitis-AI/tree/v3.5/src/vai_library/cpu_task/ops 中找到常用 OP 源代码。

**注释：**如果要实现自定义运算符的加速（PL 或 AI 引擎）函数，请将其设为 CPU 运算符，但在此 CPU 运算符的实现中实现 PL/AI 引擎调用代码。

对于步骤 4，`graph_runner` API 支持 `C++` 和 `Python`。使用 Graph_runner API 部署自定义运算符时，其运行时已最优化，包括在不同 DPU 运算符与 CPU 运算符之间采用零复制技术。它表示在不同层之间无需数据复制即可达成地址共享。

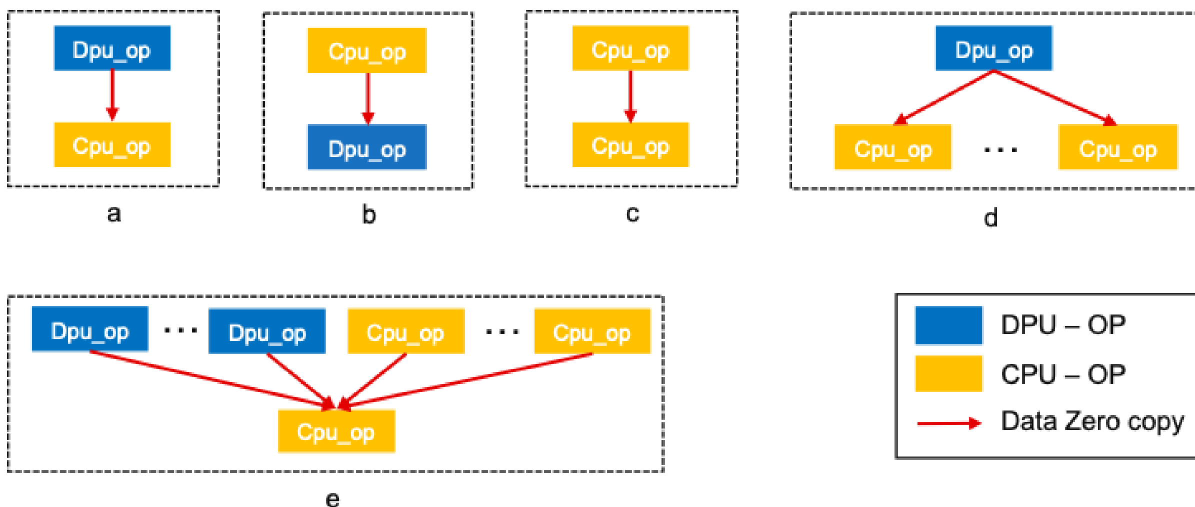
零复制技术支持下列模型结构。

表 32：**零复制技术支持的模型结构**

| 类型 | OP 输出 | OP 输入 | 使用零复制 |
|---|---|---|---|
| a | 单 dpu OP | 单 cpu OP | 是 |
| b | 单 cpu OP | 单 dpu OP | 是 |
| c | 单 cpu OP | 单 cpu OP | 是 |
| d | 单 dpu OP | 多 cpu OP | 是 |
| e | 多 cpu OP 和多 dpu OP | 单 cpu OP | 是 |

**注释：**模型结构类型 a-e 如下图所示。

图 24：**模型结构类型**




**注释：**零复制技术能否应用于其他模型结构则视情况而定。

以下提供了分别对应于两个模型的示例。

- 基于 Tensorflow2 的 MNIST 模型
- 基于 PyTorch 的 Pointpillars 模型

## 自定义运算符工作流程快速启动

本节使用含自定义运算符的 TensorFlow2 模型来演示在边缘 ZCU102 平台上快速启动自定义运算符工作流程的方式。

### 量化

1. 启动 Docker 镜像：

```
[Host]$ cd Vitis-AI
[Host]$ ./docker_run.sh xilinx/vitis-ai-cpu:latest
```

2. 下载模型源代码包 tf2_custom_op_demo.tar.gz：

```
[Docker]$ wget https://www.xilinx.com/bin/public/openDownload?
filename=tf2_custom_op_demo.tar.gz -O tf2_custom_op_demo.tar.gz
[Docker]$ tar -xzvf tf2_custom_op_demo.tar.gz
[Docker]$ cd tf2_custom_op_demo
```

3. 量化：

```
[Docker]$ conda activate vitis-ai-tensorflow2
[Docker]$ bash 1_run_train.sh
[Docker]$ bash 3_run_quantize.sh
```

量化后，./quantized/ 目录下会生成名为 quantized.h5 的量化模型。

### 编译

```
[Docker]$ vai_c_tensorflow2 -m ./quantized/quantized.h5 -a /opt/vitis_ai/
compiler/arch/DPUCZDX8G/ZCU102/arch.json -o ./ -n tf2_custom_op
```

### OP 寄存

1. 将 tensorflow2_example 文件夹复制到目标板：

   ```
   [Host]$ scp -r Vitis-AI/examples/custom_operator/tensorflow2_example
   root@[BOARD_IP]:~
   ```

2. 运行以下命令以在目标上寄存自定义运算：

   ```
   [Target]# cd ~/tensorflow2_example/op_registration/cpp
   [Target]# bash op_registration.sh
   ```

### 部署

1. 将已编译的模型复制到开发板上：

   ```
   [Host]$ scp tf2_custom_op.xmodel root@[BOARD_IP]:~
   ```

2. 下载测试图 sample.jpg 并将其复制到开发板上。
3. 在目标上编译应用代码：

   ```
   [Target]# cd ~/tensorflow2_example/deployment/cpp
   [Target]# bash build.sh
   ```

4. 运行演示：

   ```
   [Target]# ./tf2_custom_op_graph_runner ~/tf2_custom_op.xmodel ~/
   sample.jpg
   ```

## Tensorflow2 自定义运算符模型示例

以 Tensorflow2 模型为例，从此处下载代码包。参考代码包中的 the readme.md 来生成和量化模型。

### 量化模型

Tensorflow 2 提供了许多用于构建机器学习模型的常用内置层，也提供了各种简单方法，以便您轻松自行编写特定于应用的层，既可以从头开始编写，也能与现有层组合使用。层是 tf.keras 中的核心抽象之一。Layer 的子类化是创建自定义层的推荐方法。如需了解更多信息，请参阅 TensorFlow 用户指南。

Vai_q_tensorflow2 通过子类化来为新定制层提供支持。本教程将逐步演示如何量化具有自定义运算的模型。

**注释：**vai_q_tensorflow2 尚未支持通过 tf.keras.T 子类化来生成定制模型。请将其扁平化为层。

**1.训练定制层模型**

在 `train_eval.py` 中，本例定义了名为 `MyLayer` 的定制层来执行 PReLU 函数。此定制层利用可训练的权重 `alpha` 来执行以下函数。

```
f(x) = alpha * x , if x < 0
f(x) = x , if x >= 0
```

其中，`alpha` 是已习得的阵列，其形状与 `x` 相同。

随后，构建了 CNN 模型用于对 MNIST 数据集进行分类，作为示例。训练和量化模型前，请启动 Docker 并激活 `the vitis-ai-tensorflow2` 环境。运行 `1_run_train.sh` 来训练模型即可得到浮点模型 `my_model.h5`。模型准确性应 >90%。

```
bash 1_run_train.sh
```

```
Epoch 9/10
32/32 [==============================] - 0s 5ms/step - loss: 0.0108 - accuracy: 0.9960
Epoch 10/10
32/32 [==============================] - 0s 4ms/step - loss: 0.0078 - accuracy: 0.9990
313/313 [==============================] - 1s 3ms/step - loss: 0.0530 - accuracy: 0.9237

***************** Summary *****************
Trained float model accuracy:  0.9236999750137329
Trained float model is saved in  ./my_model.h5
```

此浮点模型包含了模型结构和权重，以及一个名为 `custom_layer` 的定制层。您可以从打印的摘要中获取此信息。

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 28, 28, 1)]       0
_________________________________________________________________
conv2d (Conv2D)              (None, 22, 22, 32)        1600
_________________________________________________________________
batch_normalization (BatchNo (None, 22, 22, 32)        128
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 16, 16, 32)        50208
_________________________________________________________________
batch_normalization_1 (Batch (None, 16, 16, 32)        128
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 8, 8, 32)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 4, 4, 64)          51264
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 2, 2, 64)          0
_________________________________________________________________
custom_layer (Mylayer)       (None, 2, 2, 64)          256
_________________________________________________________________
flatten (Flatten)            (None, 256)               0
_________________________________________________________________
dense (Dense)                (None, 10)                2570
=================================================================
Total params: 106,154
Trainable params: 106,026
Non-trainable params: 128
_________________________________________________________________
```

**2.（可选）评估浮点模型**

您可以运行 `2_run_eval_float.sh` 脚本来测试经过训练的浮点模型。

```
bash 2_run_eval_float.sh
```

**3.量化浮点模型**

您可以利用 vai_q_tensorflow2 `quantize_model` API 来量化带有定制层的浮点模型。代码示例如下所示：

```
from tensorflow_model_optimization.quantization.keras import vitis_quantize
quant_model = vitis_quantize.VitisQuantizer(loaded_model,
custom_objects={'MyLayer': MyLayer}).quantize_model(calib_dataset=x_test,
add_shape_info=True)
```

量化含定制层的模型时，必须将 custom_objects 实参传递到 VitisQuantizer 类。`custom_objects` 实参是包含 `{"custom_layer_class_name":"custom_layer_class"}` 的词典。多个定制层之间应以逗号分隔。此外，对于含有定制层的模型，`add_shape_info` 也应设为 True，以便为这些定制层添加形状推断信息。

在量化期间，这些定制层将在量化模型中保持不变。运行 3_run_quantize.sh 以执行量化：

```
bash 3_run_quantize.sh
```

如果一切都运行正确，the ./quantized/ 目录下将生成名为 quantized.h5 的量化模型。此模型可以用作 xcompiler 的输入，然后部署到开发板上。

```
2022-01-06 12:43:53.428589: I tensorflow/stream_executor/cuda/cuda_dnn.cc:369] Loaded cuDNN version 8204
313/313 [==============================] - 8s 17ms/step
[VAI INFO] Quantize Calibration Done.
[VAI INFO] Start Post-Quantize Adjustment...
[VAI INFO] Post-Quantize Adjustment Done.
[VAI INFO] Quantization Finished.
[VAI INFO] Start Getting Shape Information...
[VAI INFO] Getting model layer shape information
[VAI INFO] Getting Shape Information Done.
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` wi
ll be empty until you train or evaluate the model.

***************** Summary *****************
Quantized model is saved in  ./quantized/quantized.h5
```

**4.（可选）评估量化模型**

使用 model.evaluate API 评估量化模型。请务必使用正确的损失和指标来重新编译模型，因为量化中会忽略此关键字信息。

```
quantized_model.compile(loss="binary_crossentropy", metrics=["accuracy"])
quantized_model.evaluate(x_test, y_test)
```

运行 4_run_eval_quant 评估量化模型。

```
bash 4_run_eval_quant.sh
```

可以看到，量化模型具有与浮点模型相近的精度。

```
2022-01-06 12:44:56.418449: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:185] None of the MLIR Optimizati
on Passes are enabled (registered 2)
2022-01-06 12:44:57.448941: I tensorflow/stream_executor/cuda/cuda_dnn.cc:369] Loaded cuDNN version 8204
313/313 [==============================] - 3s 4ms/step - loss: 0.0897 - accuracy: 0.9207

***************** Summary *****************
Quantized model accuracy:  0.9207000136375427
```

**5.转储黄金结果**

黄金结果可用于检查数据正确性或调试已部署的模型。Vai_q_tensorflow2 提供了 dump_model API，用于转储量化模型的权重/偏差和中间激活（含样本输入）。由于 DPU 转储结果视批次而异，因此在转储黄金结果时，请将数据集的 batch_size 设为 1。

```
vitis_quantize.VitisQuantizer.dump_model( model=quant_model,
dataset=x_test[0:1], output_dir="./dump_results", dump_float=True)
```

由于定制层不进行量化，请设置 dump_float=True 为其转储浮点权重和激活。运行 5_run_dump.sh 以转储量化模型。

```
bash 5_run_dump.sh
```

您可在 ./dump_results 文件夹中查看生成的黄金结果。./dump_results/dump_results_weights 是保存的权重，而 ./dump_results/dump_results_0 则是保存的激活，其中数字 0 代表数据集的索引。

## 编译模型

以下是适用于 TensorFlow2 的命令：

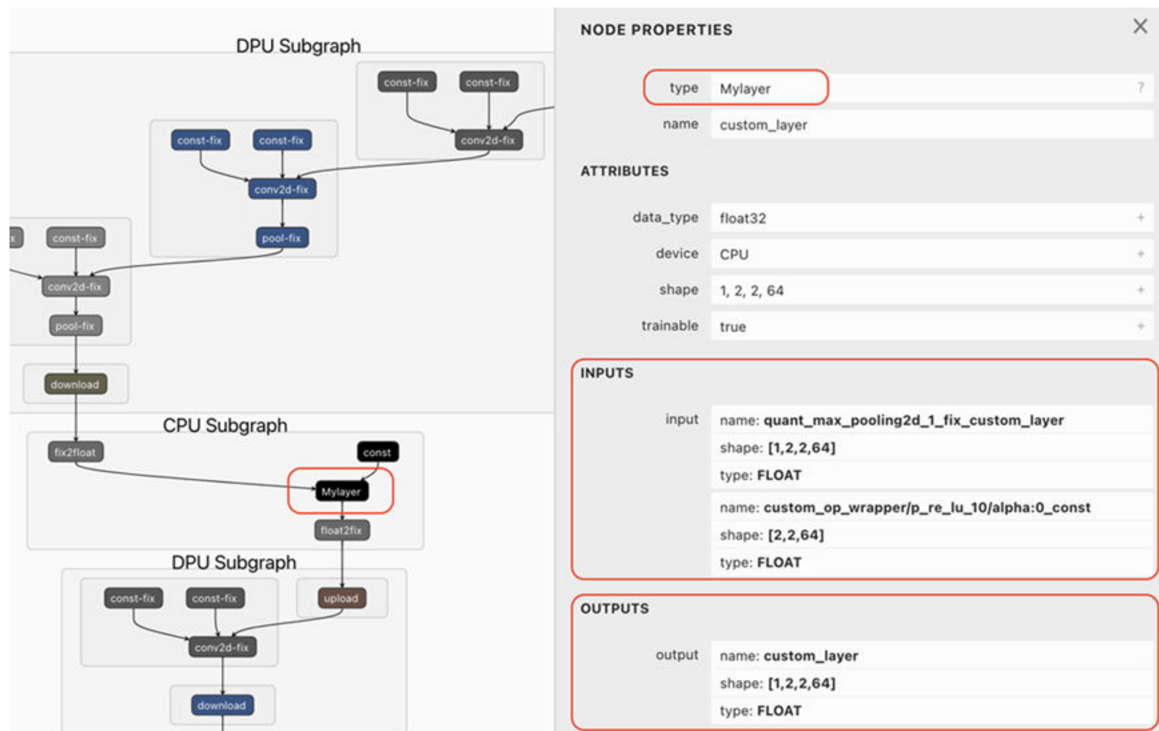
```
conda activate vitis-ai-tensorflow2
cd <path of Vitis-AI>/examples/custom_operator/tensorflow2_example/model/
quantized
vai_c_tensorflow2 -m ./quantized.h5 -a /opt/vitis_ai/compiler/arch/
DPUCV2DX8G/VEK280/arch.json -o ./ -n tf2_custom_op
```

## 自定义运算符寄存

本节逐步描述了如何寄存自定义 OP。

您可以使用 `Netron` 检查自定义 OP 属性，例如，输入和输出。

图 25：**自定义 OP 属性**




您还可使用 `xdputil` 检查 OP 的详细信息。运行以下命令检查 `custom_layer` OP：

```
xdputil xmodel tf2_custom_op.xmodel --op custom_layer
```

然后，您可以为新 OP 创建一个目录，您的所有编码和构建工作均可在这个新文件夹下完成。由于 VAI2.0 中已识别出 `Mylayer` op，请使用 `~/Vitis-AI/examples/custom_operator/tensorflow2_example/op_registration/cpp/op_Mylayer` 作为参考。

### 代码

自定义运算符寄存支持 C++ 和 Python。以下步骤描述了如何在 C++ 中实现 OP。如需了解有关如何在 Python 中实现 OP 的信息，请参阅 `Vitis-AI/examples/custom_operator/tensorflow2_example/op_registration/python/`：

1. 为新 OP 创建 CPP 文件。本例中创建的文件是 my_Mylayer_op.cpp。
2. 包含头文件 `<vart/op_imp.h>`。
3. 使用构造函数和名为 calculate 的函数来定义类。类名为 `MylayerOp in this example`。
4. 在 `calculate` 函数中实现您的算法。在此示例中实现的是 `PReLU`。
5. 通过 DEF_XIR_OP_IMP(className) 寄存您的类，对于 Mylayer，className 是 `MylayerOp`。

   **注释**：您可以从 my_Mylayer_op.cpp 复制 CPP 文件，并按需修改类名，然后即可着重完成第 4 和第 5 步。Mylayer OP 的 CPP 文件详情如下所示：

```
#include <vart/op_imp.h>
class MylayerOp {
 public:
  MylayerOp(const xir::Op* op1, xir::Attrs* attrs) : op{op1} {}
  int calculate(vart::simple_tensor_buffer_t<float> output,
                std::vector<vart::simple_tensor_buffer_t<float>> inputs) {
    CHECK_EQ(inputs.size(), 2);
    auto input_data_shape = inputs[0].tensor->get_shape();
    auto input_alpha_shape = inputs[1].tensor->get_shape();
    auto output_shape = output.tensor->get_shape();
    auto dims = output_shape.size();

    CHECK_EQ(input_data_shape.size(), 4);
    CHECK_EQ(input_alpha_shape.size(), 3);
    for (auto i = 1u; i < dims; i++)
      CHECK_EQ(input_data_shape[i], input_alpha_shape[i - 1]);

    auto element_num = inputs[0].tensor->get_element_num();
    auto alpha_size = inputs[1].tensor->get_element_num();
    for (auto i = 0; i < element_num; i++) {
      if (inputs[0].data[i] < 0) {
        output.data[i] = inputs[0].data[i] * inputs[1].data[i % alpha_size];
      } else {
        output.data[i] = inputs[0].data[i];
      }
    }

    return 0;
  }

 public:
  const xir::Op* const op;
};

DEF_XIR_OP_IMP(MylayerOp)
```

### 构建

1. 创建 Makefile 以构建 OP 库。

   请参考 `~/Vitis-AI/examples/custom_operator/tensorflow2_example/op_registration/cpp/op_Mylayer/Makefile`。
2. 设置输出目录，并添加目标（输出 .so）文件、对象 (.o) 文件和源 (.cpp) 文件之间的相依性。

   如果您在第 1 步中使用了参考 Makefile，您只需要替换文件名，包括 `libvart_op_imp_Mylayer.so`、`my_Mylayer_op.o` 和 `my_Mylayer_op.cpp`。

3. 执行“make”以完成库的构建。

   构建完成后会生成 `libvart_op_imp_Mylayer.so` 库。

Mylayer OP 的 Makefile 详情如下所示：

```
OUTPUT_DIR = $(PWD)

all: $(OUTPUT_DIR) $(OUTPUT_DIR)/libvart_op_imp_Mylayer.so

$(OUTPUT_DIR):
mkdir -p $@

$(OUTPUT_DIR)/my_Mylayer_op.o: my_Mylayer_op.cpp
$(CXX) -std=c++17 -fPIC -c -o $@ -I. -I=/install/Debug/include -Wall -
U_FORTIFY_SOURCE -D_FORTIFY_SOURCE=0 $<

$(OUTPUT_DIR)/libvart_op_imp_Mylayer.so: $(OUTPUT_DIR)/my_Mylayer_op.o
$(CXX) -Wl,--no-undefined -shared -o $@ $+ -L=/install/Debug/lib -lglog -
lvitis_ai_library-runner_helper -lvart-runner -lxir
```

**注释：**输出库名称的格式必须遵循与 libvart_op_imp_Mylayer.so 相同的格式。在此格式中，“Mylayer”是 Mylayer OP 的类型，必须改为自定义 OP 的类型。否则，在调试 OP 或运行计算图时无法链接该库。

**调试**

本节介绍了生成自定义 OP 库后如何调试自定义 OP。调试自定义 OP 前，请将 `libvart_op_imp_Mylayer.so` 复制到目标上的 `/usr/lib`。然后，在 `xdputil` 中使用 `run_op` 命令测试目标上的 OP。`run_op` 用法如下所示：

```
xdputil run_op <model_file> <op_name> [-r ref] [-d dump]
```

**注释：**对于边缘平台，请在开发板上运行 `xdputil run_op`。

执行 `xdputil run_op -h` 以查看 `run_op` 的有效实参。

```
root@xilinx-zcu102-2021_2:~# xdputil run_op -h
usage: xdputil.py run_op [-h] [-r REF_DIR] [-d DUMP_DIR] xmodel op_name

positional arguments:
  xmodel                xmodel file name
  op_name               op name, this op_name should be consistent with the name in xmodel

optional arguments:
  -h, --help            show this help message and exit
  -r REF_DIR, --ref_dir REF_DIR
                        reference directory, this directory default as "ref" should contain inputs tensor file like <TENSOR_NAME>.bin
  -d DUMP_DIR, --dump_dir DUMP_DIR
                        dump directory, this directory default as "dump" will be the dump destination of output tensor file
```

在 ref 目录中，必须提供所有输入 bin 文件，其名称应与自定义 OP 的输入张量相同。`Mylayer` 具有两个输入，其输入张量名称为 `quant_max_pooling2d_1_fix_custom_layer` 和 `custom_op_wrapper/p_re_lu_10/alpha:0_const`，如下图所示。因此，ref 目录中存储的输入文件应为 `quant_max_pooling2d_1_fix_custom_layer.bin` 和 `custom_op_wrapper_p_re_lu_10_alpha:0_const.bin`。

**注释：**执行模型量化时，可转储黄金文件。

**注释：**名称中的斜杠（“/”）应替换为下划线（“_”）。

**INPUTS**

| | |
|---|---|
| input | op_name: **quant_max_pooling2d_1_fix_custom_layer** |
| | tensor_name: **quant_max_pooling2d_1_fix_custom_layer** |
| | shape: **[1,2,2,64]** |
| | type: **FLOAT32** |
| | op_name: **custom_op_wrapper/p_re_lu_10/alpha:0_const** |
| | tensor_name: **custom_op_wrapper/p_re_lu_10/alpha:0_const** |
| | shape: **[2,2,64]** |
| | type: **FLOAT32** |

**OUTPUTS**

| | |
|---|---|
| output | op_name: **custom_layer** |
| | tensor_name: **custom_layer** |
| | shape: **[1,2,2,64]** |
| | type: **FLOAT32** |

如果仍不知道文件名，可以将输入文件放入 ref 中，然后尝试执行 run_op。然后，您即可在生成的错误信息中找到所需的文件名，如下图所示。

```
root@xilinx-zcu102-2021_2:~/Mylayer# xdputil run_op tf2_custom_op.xmodel custom_layer -r ./ref -d dump
default directory ./ref has been created, please put the input tensor file in.
root@xilinx-zcu102-2021_2:~/Mylayer# xdputil run_op tf2_custom_op.xmodel custom_layer -r ./ref -d dump
WARNING: Logging before InitGoogleLogging() is written to STDERR
W1202 04:22:48.982190 16636 tool_function.cpp:177] [UNILOG][WARNING] The operator named custom_layer, type: Mylayer, is not defined in
XIR. XIR creates the definition of this operator automatically. You should specify the shape and the data_type of the output tensor of
this operation by set_attr("shape", std::vector<int>) and set_attr("data_type", std::string)
I1202 04:22:48.987767 16636 test_op_run.cpp:79] try to test op: custom_layer
I1202 04:22:48.987846 16636 test_op_run.cpp:97]  input op: quant_max_pooling2d_1_fix_custom_layer tensor: quant_max_pooling2d_1_fix_cus
tom_layer
I1202 04:22:48.987871 16636 test_op_run.cpp:97]  input op: custom_op_wrapper/p_re_lu_10/alpha:0_const tensor: custom_op_wrapper/p_re_lu
_10/alpha:0_const
F1202 04:22:48.988036 16636 test_op_run.cpp:52] Check failed: std::ifstream(filename).read((char*)data.data, data.size).good() fail to
read! filename=./ref/quant_max_pooling2d_1_fix_custom_layer.bin;tensor=quant_max_pooling2d_1_fix_custom_layer
*** Check failure stack trace: ***
/usr/bin/xdputil: line 16: 16636 Aborted                 /usr/bin/python3 -m xdputil $*
```

成功执行 `run_op` 后（如下图所示），即可在转储目录中找到输出 bin 文件，其名称与输出张量相同。您可将其与黄金输出对比并调试自己的代码，直到两者相同为止。对于 Mylayer，输出张量名称为 `custom_layer`，因此输出 bin 文件名称应为 `custom_layer.bin`，如下图所示。

```
root@xilinx-zcu102-2021_2:~/Mylayer# xdputil run_op tf2_custom_op.xmodel custom_layer -r ./ref -d dump
WARNING: Logging before InitGoogleLogging() is written to STDERR
W1202 04:17:31.793207 16626 tool_function.cpp:177] [UNILOG][WARNING] The operator named custom_layer, type: Mylayer, is not defined in
XIR. XIR creates the definition of this operator automatically. You should specify the shape and the data_type of the output tensor of
this operation by set_attr("shape", std::vector<int>) and set_attr("data_type", std::string)
I1202 04:17:31.798681 16626 test_op_run.cpp:79] try to test op: custom_layer
I1202 04:17:31.798761 16626 test_op_run.cpp:97]  input op: quant_max_pooling2d_1_fix_custom_layer tensor: quant_max_pooling2d_1_fix_cus
tom_layer
I1202 04:17:31.798786 16626 test_op_run.cpp:97]  input op: custom_op_wrapper/p_re_lu_10/alpha:0_const tensor: custom_op_wrapper/p_re_lu
_10/alpha:0_const
I1202 04:17:31.799113 16626 test_op_run.cpp:55] read ./ref/quant_max_pooling2d_1_fix_custom_layer.bin to 0xaaab004831e0 size=1024
I1202 04:17:31.799202 16626 test_op_run.cpp:55] read ./ref/custom_op_wrapper_p_re_lu_10_alpha:0_const.bin to 0xaaab00506a90 size=1024
I1202 04:17:31.799890 16626 test_op_run.cpp:114] graph name:functional_1testing op: {
        {args: input= TensorBuffer{@0xaaab004d2fa0,tensor=xir::Tensor{name = quant_max_pooling2d_1_fix_custom_layer, type = FLOAT32, sh
ape = {1, 2, 2, 64}},location=HOST_VIRT,data=[(Virt=0xaaab004831e0, 1024)]} TensorBuffer{@0xaaab004d2220,tensor=xir::Tensor{name = cust
om_op_wrapper/p_re_lu_10/alpha:0_const, type = FLOAT32, shape = {2, 2, 64}},location=HOST_VIRT,data=[(Virt=0xaaab00506a90, 1024),(Virt=
0xaaab00506c90, 512)]}}
{
I1202 04:17:31.800941 16626 test_op_run.cpp:68] write output to dump/custom_layer.bin from 0xaaab004e11e0 size=1024
test pass
```

最后，您可将 custom_layer.bin 与自定义 OP 的黄金输出文件对比。如果两者相同，则表示实现的 OP 库正确无误。

**注释：**由于浮点数在不同的平台上会有所不同，这可能会导致转储和黄金结果不完全匹配。因此，我们提供了以下工具用于进行浮点数比较。如果差异在特定阈值内，则可将其视为一致：

```
xdputil comp_float <golden_file> <dump_file> [-t threshold] [--verbose]

-t: threshold, the default value is 0.5, in %
```

## 部署

本节介绍如何在 graph_runner API 中部署具有自定义运算符的 tensorflow2 模型。graph_runner API 支持 C++ 和 Python。如需获取 C++ 示例，请参阅 Vitis-AI/examples/custom_operator/tensorflow2_example/deployment/cpp。如需获取 Python 示例，请参阅 Vitis-AI/examples/custom_operator/tensorflow2_example/deployment/python。

**提示：** 您可以为应用创建一个新文件夹，然后将所有代码和构建文件都置于该文件夹下。

### 代码

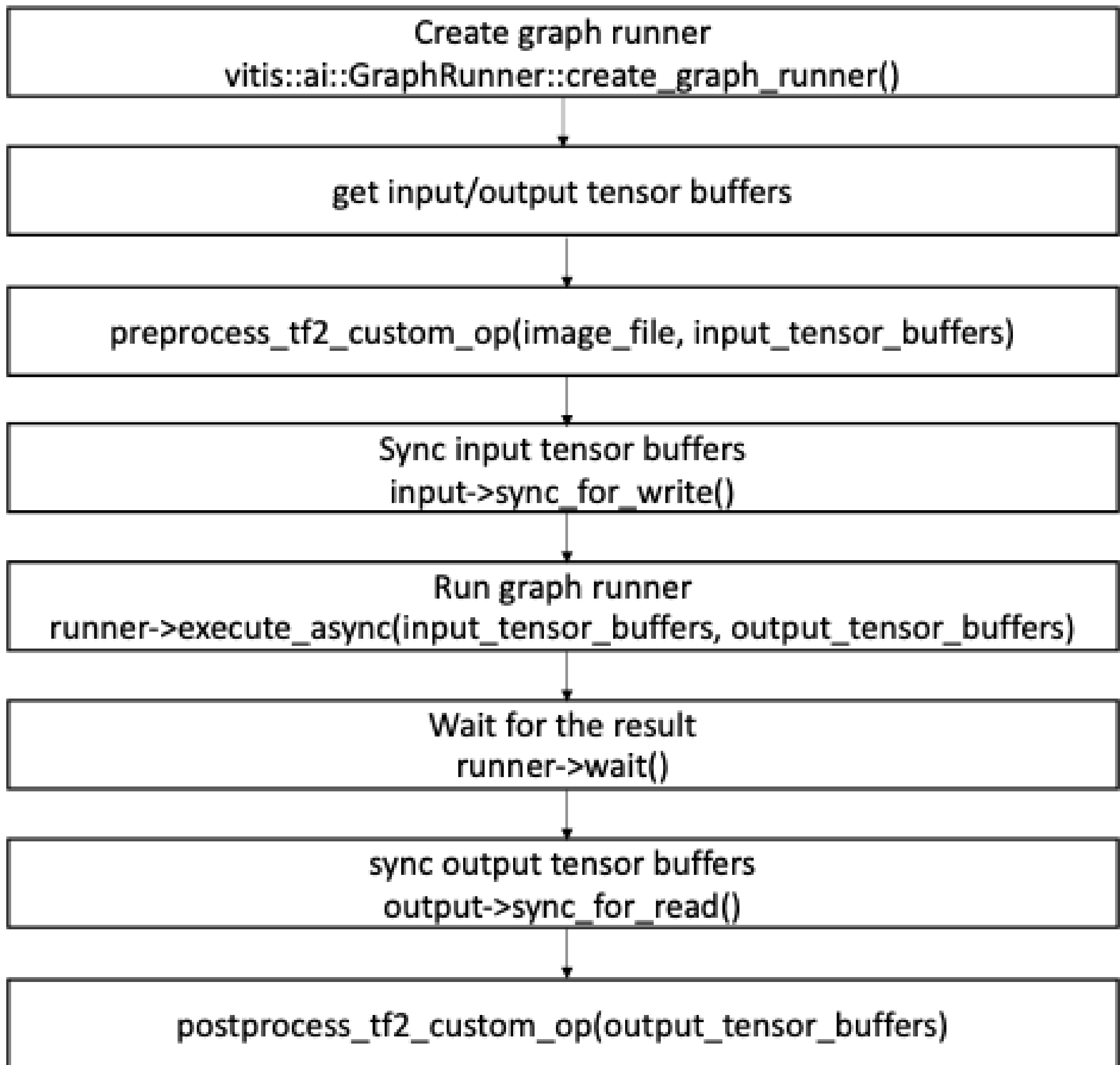
1. 创建 CPP 源文件，例如 tf2_custom_op_graph_runner.cpp
2. 包含头文件：<vitis/ai/graph_runner.hpp>
3. 在下列函数中实现模型的预处理、后处理和结果显示过程。

   ```
   static void preprocess_tf2_custom_op(...)
   static void postprocess_tf2_custom_op(...)
   static void print_result(...)
   ```

   提示：您可以从 tf2_custom_layer_graph_runner.cpp 复制 CPP 源文件，并重点处理第 3 步以实现前述三个函数。

   以下显示了 tf2_custom_op_graph_runner.cpp 的流程

图 26：**tf2_custom_op_graph_runner** 的工作流程




部分代码如下所示。

```
  // create graph runner
  auto graph = xir::Graph::deserialize(xmodel_file);
  auto attrs = xir::Attrs::create();
  auto runner =
      vitis::ai::GraphRunner::create_graph_runner(graph.get(),
attrs.get());
  CHECK(runner != nullptr);

  // get input/output tensor buffers
  auto input_tensor_buffers = runner->get_inputs();
  auto output_tensor_buffers = runner->get_outputs();

  // preprocess
  preprocess_tf2_custom_op(image_file, input_tensor_buffers);
```

```
  // sync input tensor buffers
  for (auto& input : input_tensor_buffers) {
    input->sync_for_write(0, input->get_tensor()->get_data_size() /
                             input->get_tensor()->get_shape()[0]);
  }

  // run graph runner
  auto v = runner->execute_async(input_tensor_buffers,
output_tensor_buffers);
  auto status = runner->wait((int)v.first, -1);
  CHECK_EQ(status, 0) << "failed to run the graph";

  // sync output tensor buffers
  for (auto output : output_tensor_buffers) {
    output->sync_for_read(0, output->get_tensor()->get_data_size() /
                             output->get_tensor()->get_shape()[0]);
  }

  // postprocess
  postprocess_tf2_custom_op(output_tensor_buffers);
```

**构建**

1. 遵循以下指示信息创建 `a build.sh` 以构建代码。请参阅 `Vitis-AI/examples/custom_operator/tensorflow2_example/deployment/cpp/build.sh`：

```
result=0 && pkg-config --list-all | grep opencv4 && result=1
if [ $result -eq 1 ]; then
    OPENCV_FLAGS=$(pkg-config --cflags --libs-only-L opencv4)
else
    OPENCV_FLAGS=$(pkg-config --cflags --libs-only-L opencv)
fi

CXX=${CXX:-g++}
$CXX -std=c++17 -O2 -I. \
    -o tf2_custom_op_graph_runner \
    tf2_custom_op_graph_runner.cpp \
    -lglog \
    -lxir \
    -lvart-runner \
    -lvitis_ai_library-graph_runner \
    ${OPENCV_FLAGS} \
    -lopencv_core \
    -lopencv_imgcodecs \
    -lopencv_imgproc
```

2. 在目标上执行 `bash build.sh` 以构建程序，这将生成可执行程序 `tf2_custom_op_graph_runner`。

**运行**

在运行演示前，请确保已正确设置开发板环境。另外，确保已生成或准备好以下文件。然后，将其复制到目标：

- 编译的模型，例如 tf2_custom_op.xmodel
- 自定义运算符库，例如 libvart_op_imp_Mylayer.so
- 测试来自此处的镜像
- 可执行程序，例如 tf2_custom_op_graph_runner

将自定义运算符库复制到目标上的 `/usr/lib`。然后，在目标上运行以下命令以测试模型。

```
./tf2_custom_op_graph_runner tf2_custom_op.xmodel sample.jpg
```

下图显示了运行 `tf2_custom_op_graph_runner` 的结果。

图 27：**tf2_custom_op_graph_runner** 示例

```
root@xilinx-zcu104-2021_1:~/tf2_custom_op_demo# ./tf2_custom_op_graph_runner tf2_custom_op.xmodel sample.jpg
model_file: tf2_custom_op.xmodel
image_file: sample.jpg
WARNING: Logging before InitGoogleLogging() is written to STDERR
W1128 14:12:55.312837 27054 tool_function.cpp:177] [UNILOG][WARNING] The operator named custom_layer, type: M
ylayer, is not defined in XIR. XIR creates the definition of this operator automatically. You should specify
the shape and the data_type of the output tensor of this operation by set_attr("shape", std::vector<int>) and
 set_attr("data_type", std::string)
score[0]    =  0
score[1]    =  0
score[2]    =  0
score[3]    =  0
score[4]    =  0
score[5]    =  0
score[6]    =  0
score[7]    =  0.992188
score[8]    =  0
score[9]    =  0
```

# PyTorch 自定义运算符模型示例

以 Pointpillars 模型为例，从此处下载浮点模型和代码包，然后参阅此包中的 `README.md` 来设计环境。

## 量化模型

vai_q_pytorch 提供了一个修饰器，可将一项或一组运算寄存为对 XIR 未知的自定义运算。

```
# Decorator API
def register_custom_op(op_type: str, attrs_list: Optional[List[str]] =
None):
  """The decorator is used to register the function as a custom operation.
  Args:
  op_type(str) - the operator type registered into quantizer.
  The type should not conflict with pytorch_nndct

  attrs_list(Optional[List[str]], optional) -
  the name list of attributes that define operation flavor.
  For example, Convolution operation has such attributes as padding,
dilation, stride and groups.
  The order of names in attrs_list should be consistent with that of the
arguments list.
  Default: None

"""
```

要使用自定义运算符来量化模型，请遵循以下步骤编辑代码：

1. 将目标代码移入函数中，并对其调用进行相应更改。对于 Pointpillar 模型，请将 PointPillarsScatter 模型替换为 PPScatterV2 函数。检查 `the code/test/models/voxelnet.py` 文件中的相关代码。
2. 利用修饰器 API 修饰此函数：

```
from pytorch_nndct.utils import register_custom_op
...

@register_custom_op("PPScatterV2", attrs_list=['ny', 'nx', 'nchannels'])
def PPScatterV2(ctx, voxel_features, coords, ny, nx, nchannels):
    '''
    input:
    voxel_features: B x 64 x 12000 x 1
    coords: B x 12000 x 4, 4 channels: [batch_idx, z_idx, y_idx, x_idx]
    '''
    batch_size = voxel_features.shape[0]
    # batch_canvas will be the final output.
    batch_canvas = []

    for b_idx in range(batch_size):
        # Create the canvas for this sample
        canvas = torch.zeros(nchannels, nx * ny,
dtype=voxel_features.dtype,
                             device=voxel_features.device)
        # Only include non-empty pillars

        batch_mask = coords[b_idx, :, 0] > -1
        this_coords = coords[b_idx, batch_mask, :]
        indices = this_coords[:, 2] * nx + this_coords[:, 3]
        indices = indices.type(torch.long)

        voxels = voxel_features[b_idx, :, batch_mask, 0]

        # Now scatter the blob back to the canvas.
        canvas[:, indices] = voxels
        # Append to a list for later stacking.
        batch_canvas.append(canvas)

    # Stack to 3-dim tensor (batch-size, nchannels, nrows*ncols)
    batch_canvas = torch.stack(batch_canvas, 0)
    # Undo the column sthe the tacking to final 4-dim tensor
    batch_canvas = batch_canvas.view(batch_size, nchannels, ny, nx)
    return batch_canvas
```

完成目标自定义运算符代码的准备和修饰工作后，添加常规的 vai_q_pytorch API 函数（检查 code/test/test.py 中的相关代码）

```
if quant_mode != 'float':
    max_voxel_num = config.eval_input_reader.max_number_of_voxels
    max_point_num_per_voxel =
model_cfg.voxel_generator.max_number_of_points_per_voxel
    aug_voxels = torch.randn((1, 4, max_voxel_num,
max_point_num_per_voxel)).to(device)
    # coors = torch.randn((max_voxel_num, 4)).to(device)
    coors = torch.randn((1, max_voxel_num, 4)).to(device)
    quantizer = torch_quantizer(quant_mode=quant_mode,
                                module=net,
                                input_args=(aug_voxels, coors),
                                output_dir=output_dir,
                                device=device,
                                )
    net = quantizer.quant_model
```

```
...
...
for example in iter(eval_dataloader):
...
    if quant_mode == 'test' and args.dump_xmodel:
        quantizer.export_xmodel(output_dir=output_dir, deploy_check=True)
        sys.exit()
...
...
if quant_mode == 'calib':
    quantizer.export_quant_config()
```

准备好所有更改后，运行 code/test/run_quant.sh 脚本以获取量化结果文件，包括编译器的 XMODEL 文件 (./quantized/VoxelNet_int.xmodel)：

```
sh ./code/test/run_quant.sh
```

## 编译模型

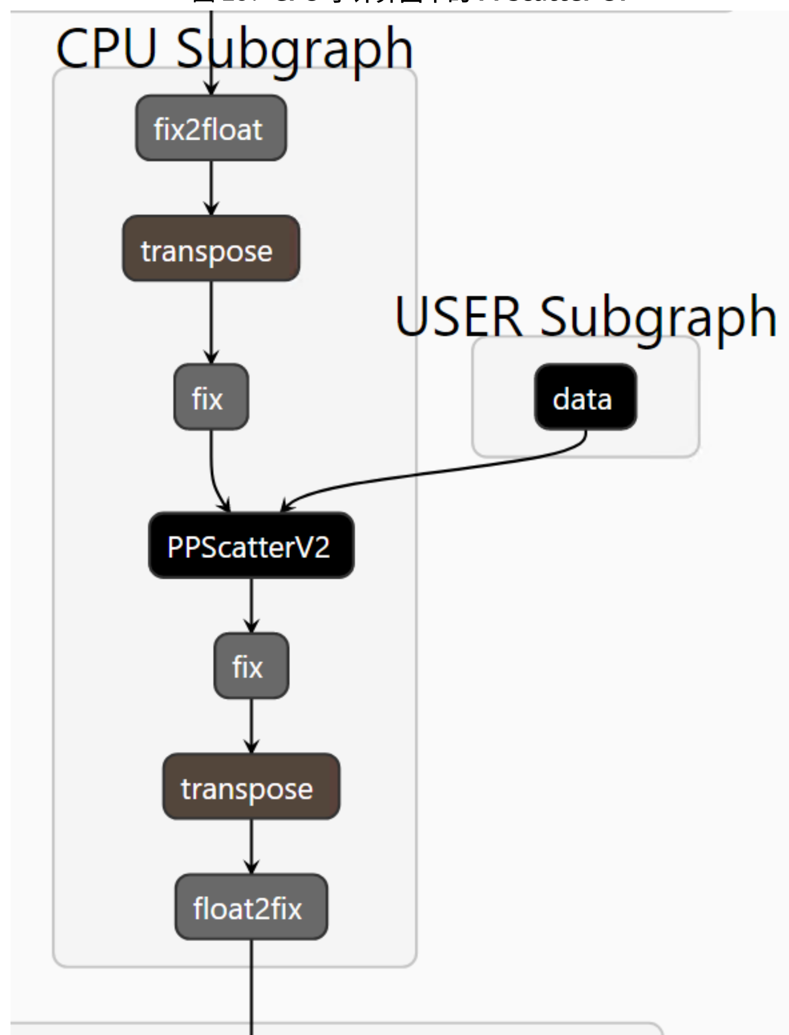
以下命令适用于 PyTorch：

```
conda activate vitis-ai-pytorch
cd <path of Vitis-AI>/examples/custom_operator/pytorch_example/model/
quantized
vai_c_xir -x VoxelNet_int.xmodel -a /opt/vitis_ai/compiler/arch/DPUCZDX8G/
ZCU102/arch.json -o ./ -n pointpillars_custom_op
```

## 自定义运算符寄存

执行自定义运算符寄存前，可使用最新 Netron 程序来检查编译的模型。在以下计算图中，PPScatter 分配给 CPU。您必须实现并寄存 `PPScatter` OP。

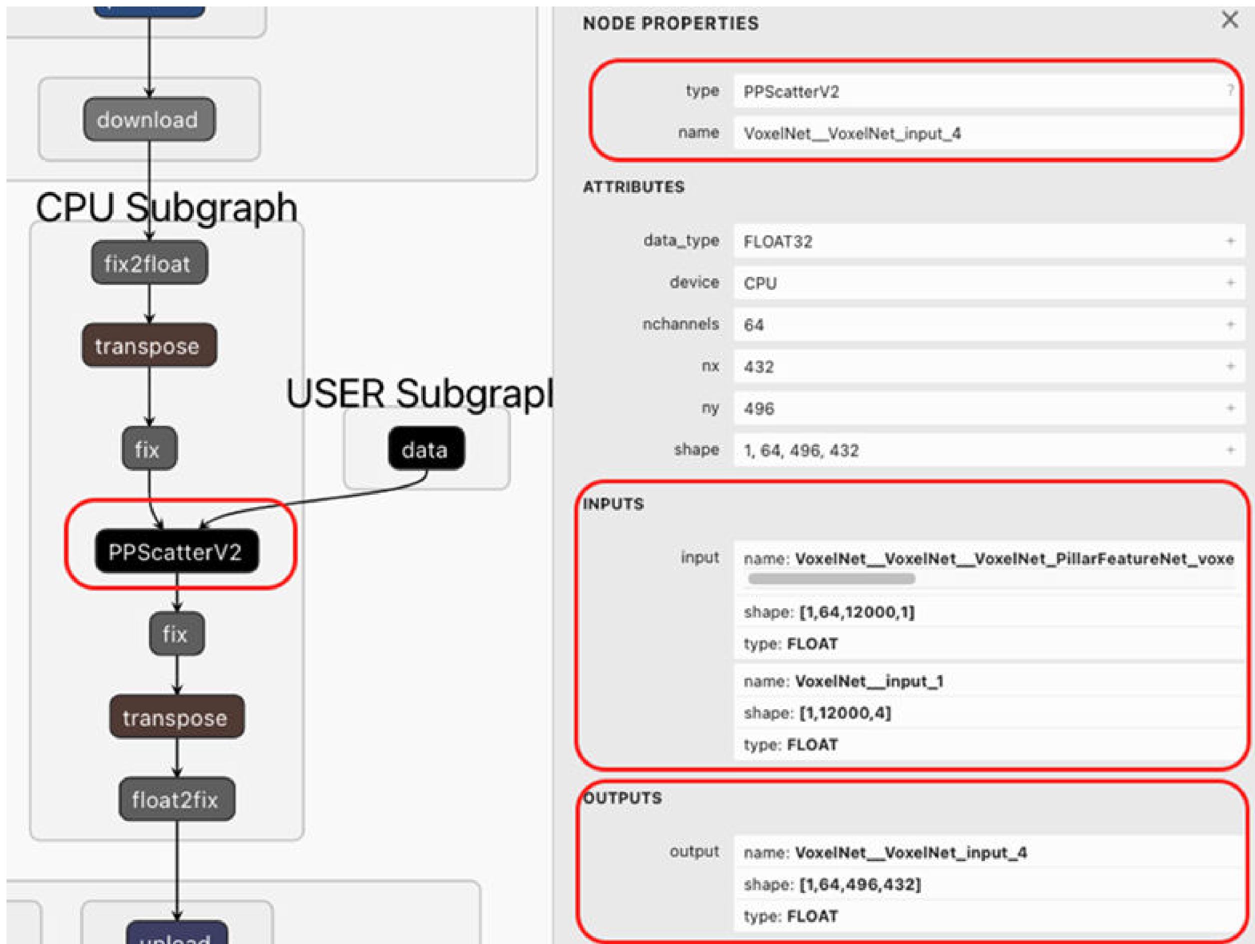
图 28：**CPU 子计算图中的 PPScatter OP**




**步骤**

1. 使用 `Netron` 打开编译的模型，并利用运算符信息查找 CPU 子计算图中的自定义 OP。

图 29：**PPScatter OP 的输入和输出**




从先前模型结构图中可以观察发现，运算类型 (op type) 为 PPScatterV2。PPScatterV2 表示要创建的自定义运算 (op) 的名称。

要获取有关自定义 OP 的详细信息，可使用 xdputil 工具。运行以下命令检查 custom_layer OP：

```
xdputil xmodel pointpillars_custom_op.xmodel --op
VoxelNet__VoxelNet_input_4
```

2. 自行编写此 op 的实现。

   自定义 OP 寄存支持 C++ 和 Python。以下步骤显示了如何在 C++ 中实现 OP。如需了解有关 OP Python 实现的信息，请参阅 `Vitis-AI/examples/custom_operator/pytorch_example/op_registration/python/`

   **注释：**在 `Vitis-AI/examples/custom_operator/op_add` 中可以找到“README.md”文件，此文件全面描述了自定义 OP 的实现过程。如需获取有关实现和寄存自定义 OP 的指导信息和说明，请参阅此“README.md”文件。

   a. 创建 `my_PPScatter_op.cpp` 源文件，并将其置于新的 `op_PPScatter` 文件夹内。

      您也可以复制现有 OP，并将其重命名为自己 OP，然后将 `my_tanh_op.cpp` 重命名为 `my_PPScatter_op.cpp`。

      ```
      cp - r Vitis-AI/src/vai_library/cpu_task/examples/op_tanh/
      op_PPScatter
      ```

b. 创建 Makefile。

```
OUTPUT_DIR = $(PWD)

all: $(OUTPUT_DIR) $(OUTPUT_DIR)/libvart_op_imp_PPScatterV2.so

$(OUTPUT_DIR):
mkdir -p $@

$(OUTPUT_DIR)/my_PPScatter_op.o: my_PPScatter_op.cpp
$(CXX) -std=c++17 -fPIC -c -o $@ -I. -I=/install/Debug/include -Wall -
U_FORTIFY_SOURCE -D_FORTIFY_SOURCE=0 $<

$(OUTPUT_DIR)/libvart_op_imp_PPScatterV2.so: $(OUTPUT_DIR)/
my_PPScatter_op.o
$(CXX) -Wl,--no-undefined -shared -o $@ $+ -L=/install/Debug/lib -
lglog -lvitis_ai_library-runner_helper -lvart-runner -lxir
```

c. 编写 op 的实现。

在 `my_PPScatter_op.cpp` 文件中，使用构造函数初始化任何必需的变量。在本例中，无需初始化任何变量。

在“calculate()”函数中实现定制逻辑。此逻辑的主要目标是从 `inputs` 变量检索输入数据、执行必要的计算，并将输出数据写入 `output` 变量。

在 calculate() 函数中，实现您自己的逻辑。该逻辑主要用于从“输入”变量获取输入数据，计算逻辑，并将输出数据写入“输出”变量。

`my_PPScatter_op.cpp` 代码如下

```
#include <vart/op_imp.h>

class MyPPScatterOp {
  public:
  MyPPScatterOp(const xir::Op* op1, xir::Attrs* attrs) : op{op1} {
  // op and attrs is not in use.
}

int calculate(vart::simple_tensor_buffer_t output,
               std::vector<vart::simple_tensor_buffer_t<float>>
inputs) {
  CHECK_EQ(inputs.size(), 2);
  auto input_data_shape = inputs[0].tensor->get_shape();
  auto input_coord_shape = inputs[1].tensor->get_shape();
  auto output_shape = output.tensor->get_shape();
  CHECK_EQ(input_data_shape.size(), 4); // 1 12000 1 64 --> 1 64
12000 1
  CHECK_EQ(input_coord_shape.size(), 3); // 1 12000 4
  CHECK_EQ(output_shape.size(), 4); // 1 496 432 64 ---> 1 64 496 432

  auto coord_numbers = input_coord_shape[1];
  auto coord_channel = input_coord_shape[2];
  CHECK_EQ(coord_numbers, input_data_shape[2]);

  auto batch = output_shape[0];
  auto height = output_shape[2];
  auto width = output_shape[3];
  auto channel = output_shape[1];
  CHECK_EQ(input_data_shape[0], batch);
  CHECK_EQ(channel, input_data_shape[1]);
```

```
  auto output_idx = 0;
  auto input_idx = 0;
  auto x_idx = 0;

  memset(output.data, 0,
output_shape[0]*output_shape[1]*output_shape[2]*output_shape[3]*sizeo
f(float));

  for (auto n = 0; n < coord_numbers; n++) {
    auto x = (int)inputs[1].data[x_idx + 3];
    auto y = (int)inputs[1].data[x_idx + 2];
    if (x < 0) break; // stop copy data when coord x == -1 .
    for(int i=0; i < channel; i++) {
    output_idx =i*height*width + y*width+x;
    input_idx = n+i*coord_numbers;
    output.data[output_idx] = inputs[0].data[ input_idx ];
    }
    x_idx += coord_channel;
  }
  return 0;
}


public:
  const xir::Op* const op;
};

DEF_XIR_op_IMP(MyPPScatterOp)
```

d. 构建库。目标目录为 `$(HOME)/build/custom_op/`。您可以修改 Makefile 中的路径。

   使用您提供的 Makefile 执行 make 命令时，自定义 OP 库将在以下目录中生成：`$(HOME)/build/custom_op/`。

   文件名为“libvart_op_imp_PPScatterV2.so”。

e. 将 `libvart_op_imp_PPScatterV2.so` 复制到目标上的 `/usr/lib`。

3. 在目标上验证 Op。

   a. 在 `xdputil` 中使用 `run_op` 命令来测试此 OP：

   ```
   xdputil run_op pointpillars_op.xmodel VoxelNet__VoxelNet_input_4 -r
   ref -d dump
   ```

   运行此命令前，请准备 OP 的参考输入。运行此命令后，就会生成 `VoxelNet__VoxelNet_input_4.bin` 文件。

   b. 将输出与黄金文件进行对比：

   ```
    xdputil comp_float ref/VoxelNet__VoxelNet_input_4.bin dump/
   VoxelNet__VoxelNet_input_4.bin
   ```

   如果 OP 实现成功，您将看到如下结果：

   ```
   root@xilinx-zcu102-2021_2:~/pointpillars_custom_op# xdputil comp_float ref/VoxelNet__VoxelNet_input_4.bin dump/VoxelNet__VoxelNet_input_4.bin
   float bin file comparation done.
   golden file and dump file are the same!
   ```

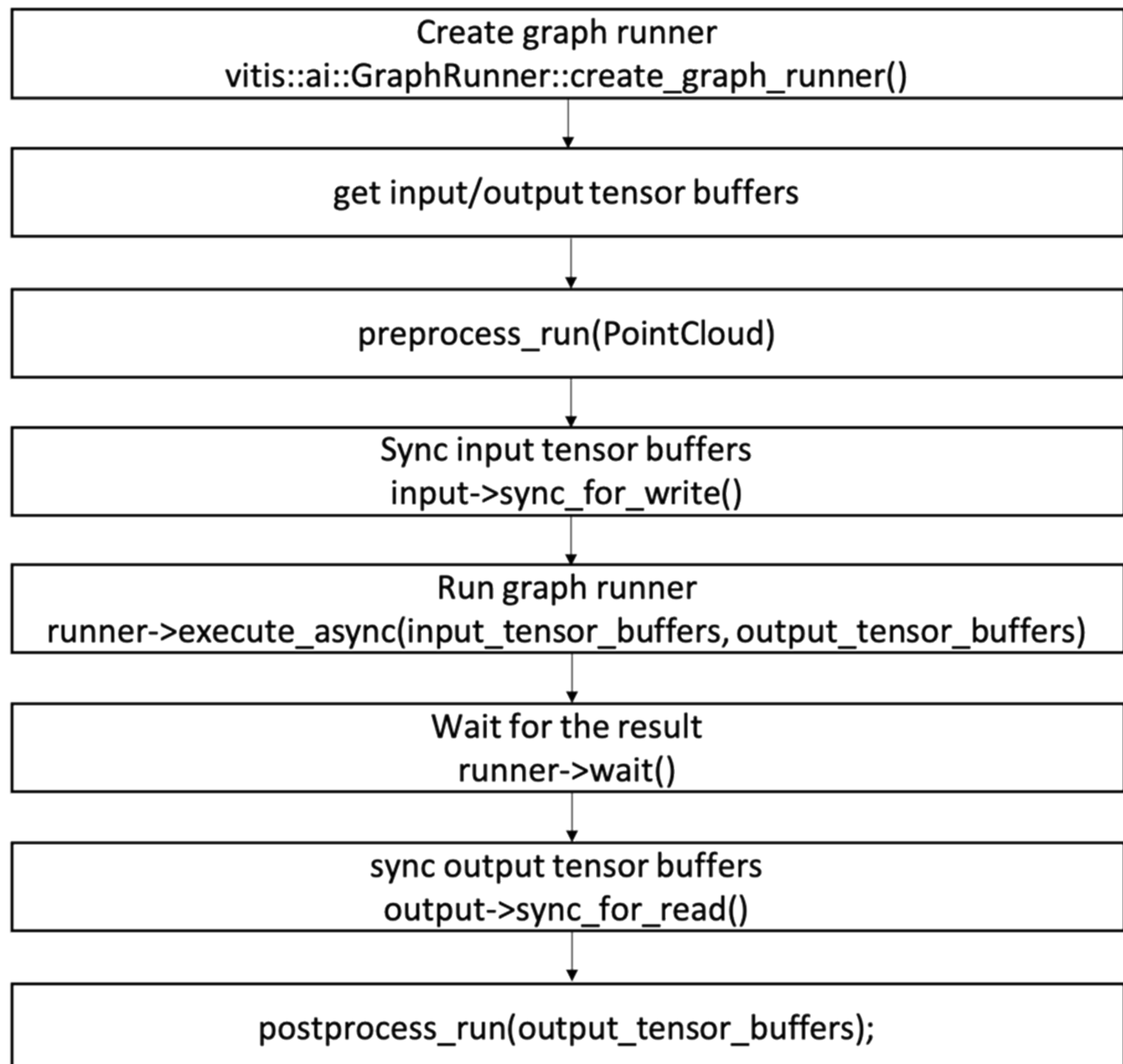
## 部署

本节描述了如何在 `graph_runner` API 中部署具有自定义运算符的 PyTorch 模型。`graph_runner` API 支持 C++ 和 Python。您可参考 `Vitis-AI/examples/vai_library/samples/graph_runner` 中的 `graph_runner` 样本：

1. 创建一个新目录来存放测试代码。
2. 为此样本创建源文件并实现以下功能：
    a. 参数解析和初始化
    b. 预处理
    c. 模型运行
    d. 后处理

    此样本的基本流程如下图所示。

图 30：**此样本的基本流程**



3. 构建程序。

   a. 如果您的工程比较简单，其中包含一个或若干个 .cpp 源文件，那么您可轻松设置构建进程，只需从 `Vitis-AI/examples/vai_library/samples/graph_runner` 复制任意现有的 build.sh 脚本，并根据具体需求对其进行调整即可。自定义 build.sh 脚本后，请执行以下命令来构建程序：

   ```
   cd <your sample folder>
   bash build.sh
   ```

   下图显示了 resnet50_graph_runner 样本的 build.sh。

   ```
   result=0 && pkg-config --list-all | grep opencv4 && result=1
   if [ $result -eq 1 ]; then
           OPENCV_FLAGS=$(pkg-config --cflags --libs-only-L opencv4)
   else
           OPENCV_FLAGS=$(pkg-config --cflags --libs-only-L opencv)
   fi

   CXX=${CXX:-g++}
   $CXX -std=c++17 -O2 -I. \
           -o resnet_v1_50_tf_graph_runner \
           resnet_v1_50_tf_graph_runner.cpp \
           -lglog \
           -lxir \
           -lvart-runner \
           -lvitis_ai_library-graph_runner \
           ${OPENCV_FLAGS} \
           -lopencv_core \
           -lopencv_imgcodecs \
           -lopencv_imgproc
   ```

   b. 如果工程比较复杂，例如该样本，那么使用 CMakeLists.txt 来轻松完成编译更好。如需了解有关 CMakeLists.txt 的更多详情，请参阅 `Vitis-AI/examples/custom_operator/pytorch_example/deployment/cpp/pointpillars_graph_runner/CMakeLists.txt`

   然后，运行以下命令以构建程序：

   ```
   cd <your sample folder>
   mkdir build
   cd build
   cmake ..
   make
   ```

   编译成功后，`<your sample folder>` 下将生成可执行程序 `sample_pointpillars_graph_runner`

4. 测试该程序。

   测试程序前，请将 XMODEL、测试图像、自定义 OP 库 `libvart_op_imp_PPScatterV2.so` 和 `sample_pointpillars_graph_runner` 可执行程序复制到开发板上。将自定义运算符库置于 `/usr/lib` 下。然后，运行以下命令以进行测试：

   ```
   ./sample_pointpillars_graph_runner ./
   pointpillars_full_customer_op.xmodel sample_pointpillars.bin
   ```

此样本的运行结果快照如下所示：

```
root@xilinx-zcu102-2021_2:~/pointpillars_graph_runner# ./
sample_pointpillars_graph_runner pointpillars_op.xmodel
sample_pointpillars.bin
WARNING: Logging before InitGoogleLogging() is written to STDERR
W1202 05:59:20.517452 1307 tool_function.cpp:177] [UNILOG][WARNING] The
operator named VoxelNet__VoxelNet_input_4, type: PPScatterV2, is not
defined in XIR. XIR creates the definition of this operator
automatically. You should specify the shape and the data_type of the
output tensor of this operation by set_attr("shape", std::vector) and
set_attr("data_type", std::string)
result: 0
0 18.541065 3.999999 -1.732742 1.703191 4.419279 1.465484 1.679375
0.880797
0 34.522400 1.505865 -1.515198 1.503061 3.550991 1.420396 1.710625
0.851953
0 10.917599 4.705865 -1.622433 1.650789 4.350764 1.634866 1.632500
0.851953
1 21.338514 -2.400001 -1.681677 0.600000 1.963422 1.784916 4.742843
0.777300
0 57.891731 -4.188268 -1.536627 1.575194 3.780010 1.512004 2.007500
0.679179
```

如果您要剖析该自定义运算符样本，请使用 DEEPHI_PROFILING=1 环境变量：

```
env DEEPHI_PROFILING=1 ./sample_pointpillars_graph_runner ./
pointpillars_full_customer_op.xmodel sample_pointpillars.bin
```

剖析结果如下所示：

```
I1130 01:29:53.038476 15571 cpu_task.cpp:163] CPU_UPDATE_INPUT : 5us
I1130 01:29:53.038684 15571 cpu_task.cpp:166] CPU_UPDATE_OUTPUT : 55us
I1130 01:29:53.038872 15571 cpu_task.cpp:169] CPU_SYNC_FOR_READ : 46us
I1130 01:29:53.039050 15571 cpu_task.cpp:181] CPU_OP_EXEC : 32us
I1130 01:29:53.039232 15571 cpu_task.cpp:181] CPU_OP_EXEC : 36us
I1130 01:29:53.039597 15571 cpu_task.cpp:181] CPU_OP_EXEC : 232us
I1130 01:29:53.066352 15571 cpu_task.cpp:181] CPU_OP_EXEC : 26575us
I1130 01:29:53.066745 15571 cpu_task.cpp:195] CPU_SYNC_FOR_WRITE : 1us
```

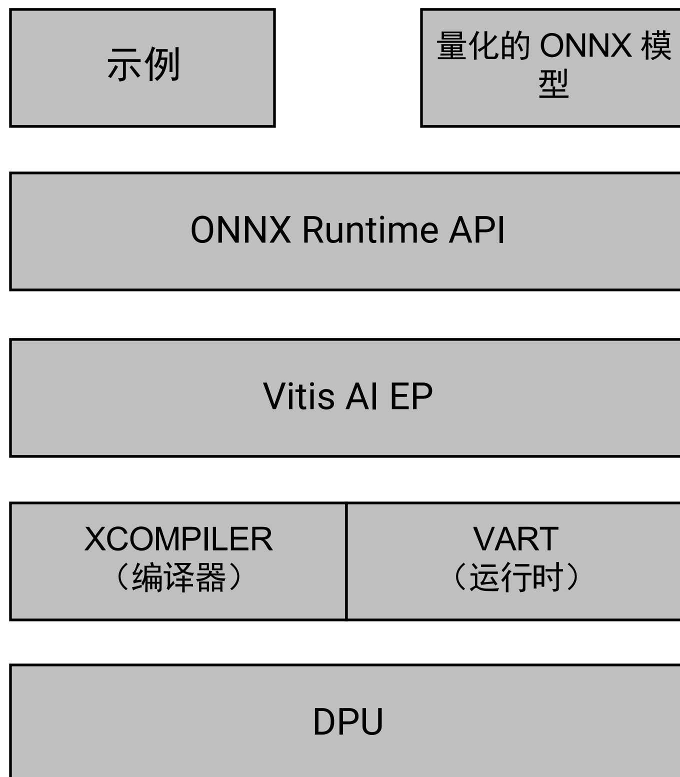
# 使用 VOE 进行编程

**ONNX 运行时**

Vitis AI Execution Provider (Vitis AI EP) 提供了含 AMD DPU 的硬件加速 AI 推断。它支持用户在目标开发板上直接运行已量化的 ONNX 模型。当前，ONNX Runtime 内的 Vitis AI EP 支持在嵌入式器件上进行神经网络模型推断加速，这类器件包括 Zynq UltraScale+ MPSoC、Versal、Versal AI Edge 和 Kria 卡。

Vitis AI ONNXRuntime Engine (VOE) 充当的是 Vitis AI EP 的实现库。

图 31：**VOE 概述**




功能特性

- 支持 ONNX Opset v18、ONNX Runtime 1.16.0 和 ONNX v1.13
- C++ 和 Python API（受 Python v3 支持）
- 除 Vitis AI EP 外，还支持结合其他执行提供程序（如 ACL EP）利用 AMD DPU 来加速推断
- 支持在 ARM64 Cortex®-A72 核上执行计算，受支持的目标为 VAI3.5 中的 VEK280

益处

- 通用性：您可在 AMD DPU 上部署子计算图，同时使用 Arm® NN 和 Arm® ACL 之类的其他执行提供程序来处理其他运算符。这种灵活性支持在目标板上部署目标板不予直接支持的模型。
- 改善性能：您可通用 AMD DPU 之类的专用执行提供程序来处理特定运算，并使用其他提供程序来处理其余运算符，从而为自己的模型实现最优化的性能。
- 扩展模型支持：增强 ONNX Runtime 即可支持部署含有 DPU 原生不支持的运算符的模型。您可通过整合其他执行提供程序来执行多种多样的模型，包括来自 ONNX model zoo 的模型。

运行时选项

Vitis AI ONNX Runtime 集成了编译器，用于将模型计算图和权重编译为微码可执行文件。该可执行文件部署在目标加速器上。

在启动 ONNX Runtime 会话期间进行模型编译，此编译进程必须在成功完成首次推断之前完成。编译时间可能不尽相同，但可能需要几分钟时间。完成模型编译后，即可缓存模型可执行文件。对于后续推断运行，您可使用缓存的可执行模型。

您可设置多个运行时变量用于配置推断会话，如下表所示。`config_file` 变量并非可选，必须将其设为指向配置文件所在位置。`cacheDir` 变量和 `cacheKey` 变量均为可选。

表 33：运行时变量

| 运行时变量 | 默认值 | 详细信息 |
|---|---|---|
| config_file | "" | （必需）表示配置文件路径，配置文件 vaip_config.json 包含在 Vitis_ai_2023.1-r3.5.0.tar.gz 内 |
| cacheDir | /tmp/{user}/vaip/.cache/ | （可选）表示高速缓存目录 |
| cacheKey | {onnx_model_md5} | （可选）表示高速缓存密钥，用于区分不同模型。 |

最终高速缓存目录为 {cacheDir}/{cacheKey}。此外，还可设置环境变量以自定义 Vitis AI EP。

表 34：环境变量

| 环境变量 | 默认值 | 详细信息 |
|---|---|---|
| XLNX_ENABLE_CACHE | 1 | 对应是否使用高速缓存，如果设为 0，则将忽略缓存的可执行文件，并且将重新编译模型。 |
| XLNX_CACHE_DIR | /tmp/$USER/vaip/.cache/{onnx_model_md5} | （可选）表示配置高速缓存路径 |

安装与部署

Vitis AI 3.5 提供了十多个基于 ONNX Runtime 的部署示例。您可在 https://github.com/Xilinx/Vitis-AI/tree/v3.5/examples/vai_library/samples_onnx 中找到这些示例。以下步骤描述了如何使用 VOE 来部署 ONNX 模型：

1. 准备 ONNX 格式的量化模型。使用 Vitis AI 量化器来量化模型，并输出 ONNX 格式的量化模型。
2. 下载 ONNX Runtime 包 vitis_ai_2023.1-r3.5.0.tar.gz 并将其安装在目标开发板上。

   ```
   tar -xzvf vitis_ai_2023.1-r3.5.0.tar.gz -C /
   ```

   然后，下载 voe-0.1.0-py3-none-any.whl 和 onnxruntime_vitisai-1.16.0-py3-none-any.whl。确保器件已联机，并在线安装这两个包。

   ```
   pip3 install voe*.whl
   pip3 install onnxruntime_vitisai*.whl
   ```

3. Vitis AI 3.5 支持 ONNX Runtime `C++` API 和 `Python` API。如需了解 ONNX Runtime API 的详细信息，请访问 https://onnxruntime.ai/docs/api/。以下提供了基于 C++ API 的 ONNX 模型部署代码片段：

   C++ 示例

   ```
   // ...
   #include <experimental_onnxruntime_cxx_api.h>
   // include user header files
   // ...

   auto onnx_model_path = "resnet50_pt.onnx"
   Ort::Env env(ORT_LOGGING_LEVEL_WARNING, "resnet50_pt");
   auto session_options = Ort::SessionOptions();

   auto options = std::unorderd_map<std::string,std::string>({});
   ```

```
options["config_file"] = "/etc/vaip_config.json";
// optional, eg: cache path : /tmp/my_cache/abcdefg // Replace abcdefg
with your model name, eg. onnx_model_md5
options["cacheDir"] = "/tmp/my_cache";
options["cacheKey"] = "abcdefg"; // Replace abcdefg with your model
name, eg. onnx_model_md5

// Create an inference session using the Vitis AI execution provider
session_options.AppendExecutionProvider("VitisAI", options);

auto session = Ort::Experimental::Session(env, model_name,
session_options);

auto input_shapes = session.GetInputShapes();
// preprocess input data
// ...

// Create input tensors and populate input data
std::vector<Ort::Value> input_tensors;
input_tensors.push_back(Ort::Experimental::Value::CreateTensor<float>(
input_data.data(), input_data.size(), input_shapes[0]));

auto output_tensors = session.Run(session.GetInputNames(), input_tensors,
session.GetOutputNames());
// postprocess output data
// ...
```

要利用 Python API，请使用以下示例作为参考：

```
import onnxruntime

# Add other imports
# ...

# Load inputs and do preprocessing
# ...

# Create an inference session using the Vitis-AI execution provider

session = onnxruntime.InferenceSession(
'[model_file].onnx',
providers=["VitisAIExecutionProvider"],
provider_options=[{"config_file":"/etc/vaip_config.json"}])

input_shape = session.get_inputs()[0].shape
input_name = session.get_inputs()[0].name

# Load inputs and do preprocessing by input_shape
input_data = [...]
result = session.run([], {input_name: input_data})
```

4. 创建 `build.sh` 文件，或者从 Vitis AI Library ONNX 示例复制该文件并对其进行修改。然后，构建该程序：

```
result=0 && pkg-config --list-all | grep opencv4 && result=1
if [ $result -eq 1 ]; then
    OPENCV_FLAGS=$(pkg-config --cflags --libs-only-L opencv4)
else
    OPENCV_FLAGS=$(pkg-config --cflags --libs-only-L opencv)
fi

lib_x=" -lglog -lunilog -lvitis_ai_library-xnnpp -lvitis_ai_library-
model_config -lprotobuf -lxrt_core -lvart-xrt-device-handle -lvaip-core -
```

```
lxcompiler-core -labsl_city -labsl_low_level_hash -lvart-dpu-controller -
lxir -lvart-util -ltarget-factory -ljson-c"
lib_onnx=" -lonnxruntime"
lib_opencv=" -lopencv_videoio -lopencv_imgcodecs -lopencv_highgui -
lopencv_imgproc -lopencv_core "

if [[ "$CXX"  == *"sysroot"* ]];then
 inc_x="-I=/usr/include/onnxruntime -I=/install/Release/include/
onnxruntime -I=/install/Release/include -I=/usr/include/xrt"
 link_x="  -L=/install/Release/lib"
else
 inc_x=" -I/usr/include/onnxruntime  -I/usr/include/xrt"
 link_x="  "
fi

name=$(basename $PWD)

CXX=${CXX:-g++}
$CXX -O2 -fno-inline -I. \
 ${inc_x} \
 ${link_x} \
 -o ${name}_onnx -std=c++17 \
 $PWD/${name}_onnx.cpp \
 ${OPENCV_FLAGS} \
 ${lib_opencv} \
 ${lib_x} \
 ${lib_onnx}
```

5. 将可执行程序和已量化的 ONNX 模型复制到目标。然后，运行该程序。

   **注释：**对于 ONNX 模型部署，输入模型是已量化的 ONNX 模型。如果环境变量 `WITH_XCOMPILER` 设为 on，那么您运行该程序时，它会首先在线执行模型编译。模型编译可能需要一段时间。

# 多 FPGA 编程

大部分现代服务器都具有多张 AMD Alveo™ 卡，您可利用这些卡来向上和向外扩展深度学习推断。Vitis AI 使用以下构建块来为多 FPGA 服务器提供支持。

## XRM

赛灵思资源管理器 (XRM) 负责管理和控制机器上的 AMD FPGA 资源。对于 Vitis AI 发行版，必须安装 XRM 才能使用 XRM 运行深度学习解决方案。XRM 是作为服务器/客户端范例来实现的。它是 XRT 基础上的附加库，旨在促进多 FPGA 资源管理。XRM 并非旨在替代 AMD XRT。XRM 的功能特性列表如下所示：

- 启用多 FPGA 异构支持
- C++ API 和 CLI 供客户端分配、使用和发布资源
- 按 FPGA、计算单元 (CU) 和服务粒度启用资源分配
- 自动发布资源
- 多客户支持：启用多客户/用户/进程请求
- XCLBIN 到 DSA 自动关联
- 客户/用户间资源共享

- 容器化支持
- 用户定义的函数
- 日志记录支持

https://github.com/Xilinx/XRM

## AI Kernel Scheduler

现实世界的深度学习应用涉及多阶段数据处理流水线，其中包含许多计算密集型预处理运算（例如，从磁盘加载数据、解码、调整大小、颜色空间转换、缩放和裁剪 CNN 等多个 ML 网络）以及 NMS 之类的各种后处理运算。

AI Kernel Scheduler (AKS) 会对对此类计算图自动进行高效流水打拍，无需用户过多干预。它可为复杂计算图的每个阶段提供不同类型的内核，这些内核可配置性高且可即插即用。部分示例包括：预处理内核（如，图像解码和大小调整）、CNN 内核（如，Vitis AI DPU 内核）以及 SoftMax 和 NMS 之类的后处理内核。您可使用内核创建其计算图，并无缝执行其作业以获得最大性能。

如需获取详细信息和示例，请参阅 Vitis AI GitHub (AI Kernel Scheduler)。

# 使用 WeGO

WeGO（全计算图优化器）是 Vitis AI 的关键功能特性，它能提供无缝解决方案用于在 Versal 数据中心 DPU 上部署 TensorFlow 和 PyTorch 模型。WeGO 通过将 Vitis AI 开发套件与 TensorFlow 和 PyTorch 框架集成，即可在 DPU 上高效部署模型。从 Vitis AI 2.5 开始，WeGO 支持 TensorFlow 2.x 和 PyTorch 1.x。它被视为是 Vitis AI 中的框架内推断解决方案，有别于使用 VART API 或 AI Library 的非框架方法。

WeGO 会自动执行子计算图分区，并为兼容数据中心 DPU 的子计算图应用最优化和加速。DPU 不支持的计算图部分（称为 CPU 子计算图）会被分派到 TensorFlow 或 PyTorch 框架，由 CPU 本机执行。WeGO 会处理包括全计算图最优化、编译和运行时子计算图分派与执行在内的整个进程。此进程完全与最终用户无关，无需最终用户参与，因此非常易用。

使用 WeGO 可为模型设计师提供从训练到推断的无缝过渡。借助 WeGO 的 Python 编程接口，即可直截了当完成在 TensorFlow 或 PyTorch 框架上部署量化模型的操作。此接口支持尽可能复用在 TensorFlow 或 PyTorch 模型训练阶段中开发的 Python 代码，包括预处理和后处理。因此，在数据中心 DPU 上部署和评估模型时，WeGO 可显著提高工作效率。

**注释：**当前，WeGO 仅支持 V70 平台上的数据中心 DPU 目标 DPUCV2DX8G。

如需获取 WeGO 示例以及有关应用 TensorFlow 和 PyTorch 来部署模型的更多信息，请参阅 Vitis AI GitHub 仓库。

## 利用 OnBoard 实现可视化

OnBoard 是 Vitis AI 3.5 版本中引入的实验性工具。其主要目的是通过可视化模型推断进程，增强 WeGO 用户的分析和调试进程。OnBoard 是作为 TensorBoard 网络服务器的扩展来构建的，通过简单的函数调用来写入事件。它能将推断结果的图像、WeGO 变换前后的模型计算图结构以及 DPU 平台相关的信息可视化。从 Vitis AI 3.5 起，OnBoard 可用于 PyTorch 和 TensorFlow 2。

如需获取 OnBoard 示例和更多信息，请参阅 Vitis AI GitHub。

# WeGO 编程接口

## PyTorch

WeGO-Torch 是 WeGO 的子工程，旨在通过将 Vitis AI 工具链集成到 PyTorch 框架内以增强 Vitis AI EoU。WeGO-Torch 遵循标准 WeGO 工作流程，有助于自动进行模型分区、编译和推断，而无需人工干预。

WeGO-Torch Python/C++ API 旨在接收通过 Vitis AI PyTorch 量化器生成的量化 TorchScript 模块作为输入。然后，它会生成一个经过优化的 TorchScript 模块，可立即用于即时推断。在 PyTorch 中利用 WeGO-Torch 加速 Vitis AI 涉及以下常规步骤：

1. 将 WeGO-Torch Python 模块导入应用。
2. 使用标准 PyTorch Python API 加载 TorchScript 量化模块。
3. 使用 WeGO-Torch 模块 API 提供 TorchScript 模块和输入形状作为输入，用于编译 TorchScript 量化模块。它会返回经最优化的 Torchscript 模块作为编译结果。
4. 使用经最优化的 TorchScript 模块运行推断。

以下代码片段显示了 WeGO-Torch 的 Python API 的基本用法：

```
import torch
# Step 1: Import WeGO-Torch python module
import wego_torch

# Step 2: Load the quantized torchscript module generated by vitis-ai
PyTorch quantizer.
model_path = <quantized_torchscript_model_path>
mod = torch.jit.load(model_path)

# Step 3: Create an optimized TorchScript module through WeGO-Torch's API.
wego_mod = wego_torch.compile(mod,
  wego_torch.CompileOptions(
  inputs_meta = [ wego_torch.InputMeta(torch.float, [1, 3, 224, 224]) ]
  )
)

# Step 4: Run inference using the optmized TorchScript module.
result = wego_mod(input)
```

在 WeGO-Torch 中使用 C++ API 编译模型的过程与此类似。有关在 WeGO-Torch 中使用 C++ API 在 DPU 上编译和运行模型的详细信息，请参阅 核 C++ 类 和 核 C++ API。

### WeGO-Torch C++ 类和 API

#### 核 C++ 类

#### wego_torch::core::CompileOptions

该 C++ 类对象用于为 WeGO-Torch 指定编译选项。您需创建该类的实例，并将其作为实参传递给 `wego_torch::core::Compile` 函数。

表 35：构造函数参数

| 类型 | 名称 | 描述 |
| --- | --- | --- |
| wego_torch::AccuracyMode | accuracy_mode | 确定精度模式。它有 2 种不同类型：<br>• wego_torch::AccuracyMode::kDefaultRemoveFixNeuron：在此模式下，WeGO-Torch 会在编译进程期间消除冗余 fixneuron 来最优化性能。这些 fixneuron 可能因量化运算符而存在，但由于板载 DPU 目标不支持这些 fixneuron，导致将其分派给 CPU 用于推断。从该模型中移除这些 fixneuron，即可增强端到端性能，前提是精度满足要求。<br>• wego_torch::AccuracyMode::kReserveFixNeuron：如果提供该值，那么 WeGO-Torch 会在模型中保留所有冗余的 fixneuron，而不是将其移除。虽然移除这些 fixneuron 能增强性能，但在某些情况下可能出现精度问题。如果使用 WeGO-Torch 编译模型后，端到端精度无法满足要求，那么建议尝试使用该值进行实验。 |
| wego_torch::core::PartitionOptions | partition_options | 设置分区选项。请参阅 wego_torch::core::PartitionOptions 类，获取更多详细信息。 |
| std::vector<InputMeta> | inputs_meta | wego_torch::InputMeta 的矢量，对应模型的每项输入。 |
| uint32_t | thread_parallel | 该参数用于最优化性能。 |
| uint32_t | core_parallel | 该参数用于最优化性能。 |
| wego_torch::core::DebugOptions | debug_options | 设置调试选项。请参阅 wego_torch::core::DebugOptions 类，获取更多详细信息。 |

**wego_torch::core::PartitionOptions**

适用于 WeGO 分区配置的选项。

表 36：**构造函数参数**

| 类型 | 名称 | 描述 |
|---|---|---|
| uint32_t | wego_subgraph_min_ops_number | 当前，WeGO 使用贪婪方法将运算符分配 DPU，前提是 DPU 兼容这些运算符。但此方法可能导致下列问题：<br>• 如果 DPU 不支持大量运算符，那么该模型最终可能会被分区到多个 DPU 子计算图和 CPU 子计算图中。如果每个 DPU 子计算图仅含少量运算符，那么在 DPU 上执行这些子计算图时，就可能因主机与器件之间存在频繁的存储器传输而导致性能问题。<br>• WeGO 会为每个 DPU 子计算图分配一个器件缓冲器。如果该模型过大并且分区后存在多个 DPU 子计算图，那么可能出现缓冲器溢出问题。<br>以下选项是专为解决此问题而添加的：wego_subgraph_min_ops_number。它会为要在 DPU 上执行的 DPU 子计算图设置最小运算符数。如不使用该选项，那么即使 DPU 支持所有运算符，也仍会在 CPU 上执行。<br><br>**注释：** wego_subgraph_min_ops_number = 0 表示无限。 |
| std::vector<std::string> | extra_accel_op_list | DPU 可运行各种 DL 运算符，但存在一些约束（例如，DPUCVDX8H_ISA1_F2W4_4PE 只能支持含 1 到 16 个内核与 1 到 4 个步幅的卷积）。WeGO 使用 DPU 限制检查引擎基于运算符的 DPU 兼容性来进行运算符分区。但某些运算符具有复杂的兼容性规则，这些规则可能导致开销过多。WeGO 默认不会将其分派给 DPU，而是由您在 extra_accel_op_list 中指定要加速的运算符。下列运算符可添加到 extra_accel_op_list 中以供 DPU 执行：<br>• aten::mul<br>• aten::mean<br>• aten::linear<br>• aten::unsqueeze<br>• aten::slice<br><br>**注释：** 如果 WeGO 在编译 extra_accel_op_list 中列出的运算符时遇到错误，则表明 DPU 无法对该特定运算符进行加速。但如未报告错误，则表示确实可以成功加速这些运算符。 |

**wego_torch::core::DebugOptions**

用于 WeGO 调试的选项。

表 37：构造函数参数

| 类型 | 名称 | 描述 |
|---|---|---|
| bool | accuracy_debug | 要为存在精度问题的子计算图启用输入和输出值的转储，请将该值设置为 true。这样即可记录这些子计算图的输入和输出。默认值为 false。 |

**wego_torch::InputMeta**

用于描述量化模型的输入的元信息。由于 Vitis AI 工具链的限制，WeGO-Torch 仅支持以静态类型和形状进行编译。您必须明确传递每个输入的日期类型和形状信息，以便 WeGO-Torch 进行类型和形状推理。

表 38：构造函数参数

| 类型 | 名称 | 描述 |
|---|---|---|
| wego_torch::DataType | type_ | 当前输入张量的数据类型。可设为：wego_torch::DataType::kBool、wego_torch::DataType::kInt32 或 wego_torch::DataType::kFloat32。 |
| wego_torch::ShapeType | input_shape_ | 当前输入张量的输入形状。 |

**wego_torch::TargetInfo**

该 C++ 类对象充当 DPU 目标信息的封装文件，可提供对板载 DPU 目标的 批次、名称、指纹和指纹驱动的信息的访问。

表 39：构造函数参数

| 类型 | 名称 | 描述 |
|---|---|---|
| std::string | name | DPU 目标的名称 |
| uint64_t | fingerprint | 板载 DPU 目标的指纹 |
| bool | is_fingerprint_driven | 指示 DPU 子计算图是基于指纹还是目标名称来编译的。 |
| uint32_t | batch | 板载 DPU 目标所支持的批次大小 |

**核 C++ API**

**wego_torch::core::Compile**

**原型设计**

```
torch::jit::Module Compile(const torch::jit::Module &module,
                           CompileOptions options);
```

表 40：参数

| 类型 | 名称 | 描述 |
|---|---|---|
| const torch::jit::Module & | module | 要编译的 PyTorch 量化模块，呈现为 torch::jit::Module 对象。 |

表 40：**参数** (续)

| 类型 | 名称 | 描述 |
|---|---|---|
| CompileOptions | options | 用于 WeGO-Torch 编译的编译器选项，指定为 wego_torch::core::CompileOptions 数据类型。如需获取有关此对象类的更多详细信息，请参阅核类章节。 |

**返回**

经最优化的 torch::jit::Module 对象。

### wego_torch::core::GetTargetInfo

**原型设计**

```
TargetInfo GetTargetInfo();
```

**参数**

无

**返回**

wego_torch::TargetInfo 对象。请参阅各类部分，以获取有关此对象类的类型的更多详细信息。

### wego_torch::getVersionInfo

**原型设计**

```
std::string getVersionInfo();
```

**参数**

无

**返回**

表示 wego_torch 版本信息的原始字符串。

## WeGO-Torch Python 类和 API

### 核 Python 类

#### wego_torch.CompileOptions(accuracy_mode : wego_torch.AccuracyMode = wego_torch.AccuracyMode.Default, inputs_meta = [], partition_options : wego_torch.PartitionOptions = None)

此 python 类对象表示 WeGO-Torch 编译选项。它由用户创建并传递到 wego_torch.compile 接口内。

表 41：**构造函数参数**

| 参数 | 描述 | 值 |
|---|---|---|
| accuracy_mode | 确定精度模式。 | • wego_torch.AccuracyMode.Default：这是精度模式的默认值。在此模式下，WeGO-Torch 在编译后会移除所有冗余的 fixneuron（修复神经元）以提升性能。存在冗余 fixneuron 的原因是因为某些运算符也一并量化。由于板载的 DPU 目标不支持这些运算符，因此会将其分派到 CPU 进行推断。只要精度能够满足所需标准，就可以从模型中移除这些 fixneuron，以提升端到端性能。<br>• wego_torch.AccuracyMode.ReserveReduantFixNeurons：如果提供该值，那么 WeGO-Torch 会在模型中保留所有冗余的 fixneuron，而不将其移除。虽然移除冗余 fixneuron 能够提升性能，但在某些情况下可能存在精度问题。如果您的模型经 WeGO-Torch 编译后无法满足端到端精度要求，那么建议您尝试提供该值。 |
| inputs_meta | 由模型的每个输入的 `wego_torch.InputMeta` 组成的列表。 | 请参阅后续章节以获取有关 wego_torch.InputMeta 类型的更多详细信息。 |
| partition_options | 分区选项，类型为 `PartitionOptions`。 | 请参阅后续章节以获取更多相关详细信息。 |

**wego_torch.InputMeta (dtype = None, input_shape = [])**

用于描述量化模型的输入的元信息。由于 Vitis AI 工具链的限制，WeGO-Torch 仅支持以静态类型和形状进行编译。您必须明确传递每个输入的日期类型和形状信息，以便 WeGO-Torch 进行类型和形状推理。

表 42：**构造函数参数**

| 参数 | 描述 |
|---|---|
| dtype | 当前输入张量的数据类型。可设为 torch.int32、torch.float 或 torch.bool。 |
| input_shape | 当前输入张量的输入形状。 |

**wego_torch.PartitionOptions (wego_subgraph_min_ops_number = 0, extra_accel_op_list = [])**

适用于 WeGO 分区配置的选项。

表 43：**构造函数参数**

| 参数 | 描述 |
|---|---|
| wego_subgraph_min_ops_number | 当前，WeGO 使用贪婪方法将运算符分派到 DPU 中，前提是 DPU 支持这些运算符。它可能会导致以下问题：<br>・ 如有大量 DPU 不支持的运算符，整个模型可能被分区为众多 DPU 子计算图和 CPU 子计算图。假设每个 DPU 子计算图仅包含少量运算符。在此情况下，子计算图仅包含少量运算符。将这些子计算图分派到 DPU 中以供执行时，就可能因主机与器件之间存在频繁的存储器传输而导致性能问题。<br>・ WeGO 会为每个 DPU 子计算图分配一个器件缓冲器。如果模型较大并且分区后存在大量 DPU 子计算图，则可能存在缓冲器上溢问题。<br>在 WeGO 中添加 `wego_subgraph_min_ops_number` 选项即可控制将 DPU 子计算图分派到 DPU 以供执行的操作。假设 DPU 子计算图中的运算符数量低于或等于 wego_subgraph_min_ops_number 阈值。在此情况下，子计算图低于或等于 `wego_subgraph_min_ops_number` 阈值。即使 DPU 可以支持子计算图中的所有运算符，WeGO 也会将该子计算图分派到 CPU 端执行。<br><br>**注释：**如果 `wego_subgraph_min_ops_number` 为 0，则无任何限制。 |
| extra_accel_op_list | DPU 可支持多种 DL 运算符，但存在某些限制（例如，DPUCVDX8H_ISA1_F2W4_4PE 只能支持含 1-16 个内核和 1-4 个步幅的卷积）。WeGO 利用 DPU 限制检查引擎在执行分区时判定某个运算符是否可受 DPU 支持。但对于部分运算符，判断运算符是否受支持的规则很复杂。为避免引发过多开销，WeGO 默认不会将其分派到 DPU 中，而是交由用户在 `extra_accel_op_list` 中显式指定需加速的运算符类型。当前可通过 `extra_accel_op_list` 来指定以下运算符以供 DPU 执行：<br>・ aten::mul<br>・ aten::mean<br>・ aten::linear<br><br>**注释：**如果在 `extra_accel_op_list` 列表中指定运算符之后，在 WeGO 中编译期间发生错误，则表示 DPU 无法对提供的运算符进行加速。否则表示可对运算符进行加速。 |

**wego_torch.TargetInfo()**

此 python 类对象旨在封装 DPU 目标信息，以便您获取板载 DPU 目标的详细信息，例如，批次大小、名称和指纹。

**注释：**建议不要手动创建该对象。您应改用 wego_torch.get_target_info() API，它会返回包含各种属性字段的预配置 wego_torch.TargetInfo 对象：

1. Batch：板载 DPU 目标所支持的批次大小。
2. Name：DPU 目标的名称。
3. Fingerprint：板载 DPU 目标的指纹。

以下是使用 wego_torch.TargetInfo 的通用方法：

```
import wego_torch
...
# Detect the DPU target on-board and return an object with type
wego_torch.TargetInfo.
target_info = wego_torch.get_target_info()
# The target_info object can be printed directly.
print(target_info)
# Retrieve diverse propery fields of the DPU target.
batch, name, fingerprint = target_info.batch, target_info.name,
target_info.fingerprint
...
```

**核 Python API**

表 44：**wego_torch.compile(module: Any, options: wego_torch.CompileOptions)**

| 描述 | 参数 | 返回 |
|---|---|---|
| 编译 PyTorch torchscript 模块以供 Vitis AI 加速。 | • module：经量化的 PyTorch 模块，其中包含要编译的 torch.jit.ScriptModule 类型。<br>• options：用于 WeGO-Torch 编译的编译器选项，含 wego_torch.CompileOptions 类型。请参阅各核类，了解有关详情。 | 经优化的 TorchScript 模块。 |

表 45：**wego_torch.get_target_info()**

| 描述 | 参数 | 返回 |
|---|---|---|
| 检测板载 DPU 目标，并返回目标信息。 | 无 | wego_torch.TargetInfo 对象。请参阅各核类，了解有关详情。 |

表 46：**wego_torch.version()**

| 描述 | 参数 | 返回 |
|---|---|---|
| 获取 WeGO-Torch 版本。 | 无 | 表示 wego_torch 版本信息的原始字符串。 |

## WeGO-PyTorch 限制

WeGO-Torch 工程当前尚处于抢先体验状态，可能存在一些已知的使用问题。以下是这些问题的详情以及解决这些问题的必要步骤：

1. WeGO-Torch 无法支持 RCNN 模型（含控制流），原因是：

    a. RCNN 模型支持：由于动态形状问题，WeGO-Torch 目前不支持含控制流的 RCNN 模型。在 RCNN 模型中，提供不同的图像作为输入时，张量的形状会在运行时发生变化。这给 WeGO 的部署带来了挑战。为使 RCNN 模型与 WeGO 兼容，需要手动修改以消除这一限制。

    b. 输入类型兼容性：RCNN 模型通常接受 `Tensor[]` 作为输入类型，而 WeGO 的编译 API 不支持这种输入类型。另外，使用 `Tensor []` 作为输入类型暗示浮点模型本身对批次敏感，在 TorchScript 追踪阶段期间使用不同批次大小时，通过追踪获取的量化模型不尽相同。要在 WeGO 中部署这些模型，建议采取以下步骤：

i. 将 `Tensor []` 替换为 `Tensor` 或 `Tensor, Tensor, ...` （当输入数量已知时）作为原始浮点模型中的输入类型。

ii. 在 WeGO 中用于推断的批次大小必须与量化期间导出阶段所使用的批次大小相同。

2. WeGO-Torch 目前只覆盖了数据中心 DPU 可以支持的一部分运算符。因此，即使数据中心的 DPU 可以支持某些运算符，它们也可能被分派到 CPU 执行。

### 示例

如需获取 WeGO-Torch 示例，请参阅 Vitis AI GitHub 页面。

## TensorFlow 2.x

WeGO-TensorFlow2.x 是 WeGO 的子工程，旨在通过将 Vitis AI 工具链无缝集成到 TensorFlow 2.x 框架中来增强 Vitis AI EoU。VAI 2.5 支持 TensorFlow v2.8.0 版本。要使用 WeGO-TensorFlow2.x，您应提供 HDF5 格式的量化模型，该模型通常命名为 quantized.h5，可以使用 vai_q_tensorflow2 量化器生成。

WeGO 的核 API create_wego_model() 会自动将量化的 Keras 模型转换为新的具体函数。该函数会将数据中心 DPU 兼容的子计算图转换为 VaiWeGOOp 类型的 TensorFlow 运算符。

WeGO-TensorFlow2.x 的整个推断进程可抽象为以下 4 个步骤：

1. 将 WeGO TensorFlow2.x Python 模块导入应用。
2. 使用 vitis_vai.get_target_info() 获取 DPU 目标的批次信息，用于输入批次进程。
3. 使用 vitis_vai.create_wego_model() 创建 WeGO 模型获取具体函数。
4. 执行该具体函数。

### WeGO-TensorFlow2.x Python API

#### 核 Python 类

#### DeviceInfo()

此对象用于封装 DPU 目标信息。

**注释：**您不应自行创建此对象，而应依靠 API Vitis_vai.get_target_info() 来返回此对象，其中包含多种属性字段：

- batch：板载 DPU 目标所支持的批次大小。
- target：DPU 目标的名称。
- fingerprint：板载 DPU 目标的指纹。

DeviceInfo 的通用用法如下：

```
from tensorflow.compiler import vitis_vai
...
target_info = vitis_ai.get_target_info()
batch = target_info.batch
name = target_info.target
fingerprint = target_info.fingerprint
...
```

### 核 Python API

表 47：**get_target_info()**

| 描述 | 参数 | 返回 |
|---|---|---|
| 获取目标信息，包括批次、指纹和目标名称。此信息可用于批处理或者获取目标名称信息。 | 无 | DeviceInfo 对象。如需了解有关此对象类型的更多详情，请参阅核的各类部分。 |

表 48：**create_wego_model(input_h5, feed_dict={}, accuracy_mode=vitis_vai.enums.AccuracyMode.Default)**

| 描述 | 参数 | 返回 |
|---|---|---|
| 创建 WeGO 模型，将 Keras h5 文件转换为具体函数 | 1. input_h5：指向 h5 文件的路径。<br>2. feed_dict：输入不含固定输入形状的模型时，用于推断形状配置。<br>3. accuracy_mode：<br>· vitis_vai.enums.AccuracyMode.Default：不含 CPU FixNeuron 情况下的推断。<br>· vitis_vai.enums.AccuracyMode.ReserveReduantFixNeurons：含 CPU FixNeruon 情况下的推断 | 含 VaiWeGOOp 的新具体函数。<br><br>**注释：**默认情况下，WeGO 通过消除量化模型中的 CPU FixNeuron 运算符来达成最优性能。但对于包含大量 CPU FixNeuron 运算符的模型，以默认值 (vitis_vai.enums.AccuracyMode.Default) 部署这些模型可能导致其精度降低。在此类情况下，您可切换至 vitis_vai.enums.AccuracyMode.ReserveReduantFixNeurons 以提升精度。 |

### 环境变量

#### WEGO_ENABLE_AGGRESSIVE_SHAPE_INFERENCE

如果某些运算符需依赖 batchsize（批次大小）来推断静态形状，则可启用此环境变量。导出 WEGO_ENABLE_AGGRESSIVE_SHAPE_INFERENCE=1 会将批次大小设为 1。例如，对于某些模型（例如，含输入形状 [-1,640,640,3] 的 ssd_resnet_50_fpn_coco_tf 模型），无法获取 reshape 运算符的静态形状，导致由 Vitis AI 工具链编译部分 WeGO 子计算图时出现错误。错误信息如下所示：

```
AssertionError: [ERROR] Invalid shape of input layer: shape: [1, -1, -1,
256] (N,H,W,C), name: input1
[INFO] parse raw model     :   0%|          | 0/52 [00:00<?, ?it/
s]
*** Check failure stack trace: ***
```

为解决此问题，必须先按如下所示来设置环境变量，然后再运行样本以启用 WeGO 积极形状推断：

```
export WEGO_ENABLE_AGGRESSIVE_SHAPE_INFERENCE=1
```

否则，您无需使用此环境变量。或者，取消以下命令设置的环境变量：

```
unset WEGO_ENABLE_AGGRESSIVE_SHAPE_INFERENCE
```

## 示例

如需获取 WeGO-TensorFlow 2.x 样本，请参阅 Vitis AI GitHub 页面。

## TensorFlow 1.x

WeGO-TensorFlow1.x 是 WeGO 的子工程，旨在通过将 Vitis AI 工具链集成到 TensorFlow 1.x 框架中来改善 Vitis AI EoU。Vitis AI 2.5 支持 TensorFlow v1.15。WeGO-TensorFlow1.x 的输入是量化模型，此量化模型通常名为 quantize_eval_model.pb，由 vai_q_tensorflow 生成。核 WeGO API create_wego_graph() 会将经量化的计算图自动转换为新的 TensorFlow 计算图，称为 WeGO 计算图，其中数据中心 DPU 兼容的子计算图会变换为含某种类型的 VaiWeGOOp 的 TensorFlow 运算符。

整个 WeGO-TensorFlow1.x 接口均可抽象化为下列步骤

1. 在原始计算图上执行某些计算图级最优化，以满足 DPU 特有要求。
2. 遍历输入量化模型的整个计算图，检测数据中心 DPU 支持的节点。
3. 根据步骤 2 中检测所得节点列表，基于量化的计算图执行计算图自动分区。
4. 将所有数据中心 DPU 兼容的子计算图都变换为新的 TensorFlow 节点，并在输入量化模型内包含某种类型的 VaiWeGOOp。
5. 返回经最优化的新 WeGO 计算图，然后调用 TensorFlow sess.run() 以执行整个计算图。

### WeGO-TensorFlow 1.x Python API

#### 核 Python 类

#### DeviceInfo()

此对象用于封装 DPU 目标信息。

**注释：**您不应自行创建此对象，而应依靠 API vitis_vai.get_target_info() 来返回此对象，其中包含多种属性字段：

- batch：板载 DPU 目标所支持的批次大小。
- target：DPU 目标的名称。
- fingerprint：板载 DPU 目标的指纹。

DeviceInfo 的通用用法如下：

```
from tensorflow.contrib import vitis_vai
...
target_info = vitis_ai.get_target_info()
batch = target_info.batch
name = target_info.target
fingerprint = target_info.fingerprint
...
```

#### 核 Python API

表 49：**get_target_info()**

| 描述 | 参数 | 返回 |
|---|---|---|
| 获取目标信息，包括批次、指纹和目标名称。此信息可用于批处理或者获取目标名称信息。 | 无 | DeviceInfo 对象。如需了解有关此对象类型的更多详情，请参阅核的各类部分。 |

表 50：**create_wego_graph(input_graph_def, feed_dict={}, accuracy_mode= vitis_vai.enums.AccuracyMode.Default)**

| 描述 | 参数 | 返回 |
|---|---|---|
| 用于 VAI 变换的 Python 封装文件。 | 1. input_graph_def：包含要变换的模型的 GraphDef 对象。<br>2. feed_dict：输入不含固定输入形状的模型时，用于推断形状配置。<br>3. accuracy_mode：<br>・ vitis_vai.enums.AccuracyMode.Default：在不含 CPU FixNeuron 的情况下运行。<br>・ vitis_vai.enums.AccuracyMode.ReserveReduantFixNeurons：在含 CPU FixNeruon 的情况下运行 | 新 GraphDef，其中所含 VaiWeGOOp 置于计算图中以替代子计算图。<br><br>**注释：**默认情况下，WeGO 通过消除量化模型中的 CPU FixNeuron 运算符来达成最优性能。但对于包含大量 CPU FixNeuron 运算符的模型，以默认值 (Vitis_vai.enums.AccuracyMode.Default) 部署这些模型的精度时，其精度可能精度降低。在此类情况下，您可切换至 Vitis_vai.enums.AccuracyMode.ReserveReduantFixNeurons 以提升精度。 |

**环境变量**

**WEGO_ENABLE_AGGRESSIVE_SHAPE_INFERENCE**

此环境变量可供 WeGO TensorFlow 1.x 和 WeGO TensorFlow 2.x 使用。请参阅 WeGO TensorFlow 2.x 章节以了解其用法。

### 示例

如需获取 WeGO-TensorFlow 1.x 样本，请参阅 Vitis AI GitHub 页面。

## WeGO 中的即时量化

在最初的 WeGO 工作流程中，由于 WeGO 只接受量化的 INT8 模型作为输入，因此最初必须执行单独的量化进程。这可通过显式使用 Vitis AI 量化器将 float32 模型转换为 INT8 模型来实现。由此导致需要为用户执行额外的任务，例如，在量化器与 WeGO 之间执行 Conda 环境切换操作，以便明确 Vitis AI 量化器与 WeGO 之间的关系。为了提升易用性，使量化到部署的整个进程更顺畅，WeGO 将 Vitis AI 量化器集成到其流程中，这样在提供 float32 模型作为 WeGO 的输入时，即可启用即时量化。除了用于编译的原始 WeGO API 之外，在 WeGO 中还引入了一个全新的 API 用于量化，并且量化器细节完全与最终用户无关，无需最终用户参与。WeGO 中的量化集成尚处于早期阶段，存在下列限制：

1. 此集成流程目前仅支持 PTQ（训练后量化）。如果模型精度与期望相去甚远，必须执行微调或者使用量化感知训练 (QAT) 遵循原生 Vitis AI 量化流程来提升精度。
2. 在 WeGO 中当前仅采用 CPU 进行量化，当前不支持 GPU。量化大型模型时，这可能引发一些问题，并且此进程可能耗用大量时间。

### 量化 API

本节介绍了 WeGO 的 API，这些 API 适用于面向不同框架的 PTQ 量化。

## PyTorch

### 量化 API

```
wego_torch.quantize(
module: torch.nn.Module,
input_shapes: Sequence[Sequence],
dataloader: Iterable,
calibrator: Callable[[torch.nn.Module, Any, int, torch.device], None],
export_dataloader: Iterable = None,
device: torch.device = torch.device("cpu"),
output_dir: str = "quantize_result",
bitwidth: int = None,
quant_config_file: Optional[str] = None,
*args, **kwargs) -> torch.jit.ScriptModule
```

此函数采用训练后量化 (Post Training Quantization, PTQ) 方法来量化 torch 浮点模型，并返回量化后的 TorchScript 模块用于 WeGO 编译。

如果 PTQ 无法达到所需精度，您可能需要考虑使用量化感知训练 (QAT) 搭配 Vitis AI 量化器 API。如需深入了解量化进程，请参阅 Vitis AI 用户指南中的 量化模型。

#### 参数

- module：(`torch.nn.Module`) 输入 PyTorch 浮点模型。
- input_shapes：(`Sequence[Sequence]` 模型的输入形状：列表或元组序列。
- dataloader：(`Iterable`) 校准数据集的数据加载器。它必须可迭代。API 会通过它进行迭代并将返回的值传递给校准器。
- calibrator：(`Callable`) 可调用对象，用于执行批量数据预处理和前传。从数据加载器获取批量数据、按需对其进行预处理，并使用模块对其进行转发。此校准器会在校准和导出阶段调用 N + 1 次。
  - 阶段 1 用于校准。在此阶段，您的数据加载器会进行迭代，并通过模块进行数据传递以收集量化统计数据。校准器会调用 N 次 (N = len(dataloader))。在阶段 1 中，如果未传递可选 export_dataloader，则会保存数据加载器返回的第一个批次，稍后供阶段 2 使用。在此情况下，请确保校准器或迭代副作用未更改第一个批次。
  - 阶段 2 用于导出量化的 TorchScript 模块。在此阶段中，校准器仅调用一次，含一个批次的数据。如果您传入 export_dataloader，那么此 export_dataloader 会迭代，并且仅使用第一个批次。程序处理完第一个批次后，会中断迭代。如果您未传入 export_dataloader，则会使用从阶段 1 保存的第一个批次。

    **校准器实参：**

    - module：(`torch.nn.Module`) 用于量化的模块。这是您传入的模块的修改版本，其中具有用于收集数据统计信息的必要机制。您应使用此模块代替原始浮点模型来进行数据前传。
    - batch_data：(`Any`) 从数据加载器返回的批次数据。
    - batch_index：(`int`) 批次索引。请按需使用器件 (torch.device) 进行前传。当前仅支持 CPU。

    **注释：** 用于量化 API 的额外定位实参和关键字实参都将前传给校准器。如需了解更多信息，请参阅 量化模型。
- export_dataloader：(`Iterable`) 可选数据加载器，用于导出阶段。默认值为 None。如果值为 None，则使用从阶段 1 保存的第一个批次。
- device：(`torch.device`) 用于校准的器件。当前仅支持 CPU。
- output_dir：(`str`) 临时工作目录。默认值为 quantize_result。部分中间文件保存在此处。

- bitwidth：（int）全局量化位宽。默认值为 8。
- quant_config_file：（str）指向量化器配置文件的路径。默认值为 None。
- args：要传递给校准器的额外定位实参。
- kwargs：要传递给校准器的额外关键字实参。

如需了解有关如何使用 WeGO 中的即时量化的更多信息，请参阅 WeGO 示例。

## TensorFlow 2.x

### 量化 API

```
vitis_vai.quantize(
input_float,
quantize_strategy = 'pof2s',
custom_quantize_strategy = None,
calib_dataset = None,
calib_steps = None,
calib_batch_size = None,
save_path = './vai_wego/quantized.h5',
verbose = 0,
add_shape_info = False,
dump = False,
dump_output_dir = './vai_dump/')
```

此函数用于执行浮点模型训练后量化 (PTQ)，包括模型最优化、权重量化和激活训练后量化。

#### 参数

- input_float：要量化的 `tf.keras.Model` 浮点对象。
- quantize_strategy：表示量化策略类型的字符串对象。可用值包括：pof2s、pof2s_tqt、fs 和 fsx。pof2s 是默认策略，使用 2 的幂值比例量化器和 Straight-Through-Estimator（直通式估算器）。pof2s_tqt 是 Vitis AI 1.4 中引入的策略，使用 2 的幂值比例量化器中经过训练的阈值，可生成更好的 QAT 结果。fs 是 Vitis AI 2.5 中引入的量化策略，它为 Conv2D、DepthwiseConv2D、Conv2DTranspose 和 Dense 层的输入和权重执行浮点比例量化。另一方面，相比于 fs 量化策略，fsx 能将量化策略扩展到更多类型的层，包括 Add、MaxPooling2D 和 AveragePooling2D，并且还在量化中引入了偏差和激活。

  **注释：**

  - pof2s_tqt 只能用于 init_quant=True 的 QAT，以获得最佳性能。
  - fs 和 fsx 策略是专为具有浮点支持的目标器件设计的。DPU 当前不支持浮点，因此以这些量化策略进行量化的模型无法部署到 DPU。

- custom_quantize_strategy：string 对象。表示定制量化策略 JSON 文件的文件路径。
- calib_dataset：tf.data.Dataset、keras.utils.Sequence 或 np.numpy 对象。用于校准的代表性数据集。您可以将 eval_dataset、train_dataset 或其他数据集整体或其中一部分用作 calib_dataset。
- calib_steps：int 对象。表示校准步骤总数。可忽略，默认值为 None。如果 calib_dataset 是 tf.data 数据集、生成器或 keras.utils.Sequence 实例且 steps 为 None，校准会运行到数据集耗尽为止。阵列输入不支持此实参。
- calib_batch_size：int 对象。表示用于校准的每批次样本数。如果 calib_dataset 为数据集、生成器或 keras.utils.Sequence 实例形式，则批次大小由数据集本身控制。如果 calib_dataset 采用 numpy.array 对象形式，那么默认批次大小设为 32。

- save_path：string 对象。量化模型的保存目录。
- verbose：int 对象。表示日志记录的详细程度。详细程度值越大，生成的 log 日志记录越详细。默认值为 0。
- add_shape_info：bool 对象。用于判定是否要为定制层添加形状推断信息。对于含有定制层的模型，其值必须设为 True。
- dump：用于启用或禁用转储的标志。如果 dump=False，则禁用转储，如果 dump=True，则启用转储。
- dump_output_dir：string 对象。表示转储结果的保存目录。

如需了解有关如何在 WeGO TensorFlow 2.x 中使用即时量化的更多信息，请参阅 WeGO 示例。

## TensorFlow 1.x

### 量化 API

```
def quantize(
input_frozen_graph = "",
input_nodes = "",
input_shapes = "",
output_nodes = "",
input_fn = "",
method = 1,
calib_iter = 100,
output_dir = "./quantize_results",
**kargs)
```

该函数会在 WeGO TensorFlow r1.15 中调用 `vai_q_tensorflow` 命令工具，并将输入浮点模型转换为定点模型，用于 DPU 部署加速。为了与原生 `vai_q_tensorflow` 量化器完全兼容，从该 API 接收到的所有参数都会直接前传到 `vai_q_tensorflow` 命令工具。该函数会返回量化的 `GraphDef` 对象，或者如果失败则返回 `None`。

**注释：**当前仅支持通过 PTQ 在 WeGO 中进行即时量化。如需了解有关快速微调和 QAT 量化的更多信息，请参阅 vai_q_tensorflow 量化感知训练。

### 参数

- input_frozen_graph：字符串。指向输入冻结计算图 (.pb) 的路径（默认值：）
- input_nodes：字符串：要量化的子计算图的输入节点名称列表（以逗号分隔），搭配 output_nodes 一起使用。生成部署模型时，仅包含 input_nodes 与 output_nodes 之间的子计算图。请将其设置为要量化的模型主体开始位置，例如，数据预处理或数据增广之后的节点。（默认值：）
- input_shapes：字符串。input_nodes 的形状列表（逗号分隔）。每个节点的形状必须是以逗号分隔的 4 维形状，例如，`1,224,224,3`；支持批次大小未知，例如，`?,224,224,3`；如有多个 input_nodes，请给每个节点的形状列表赋值，以 `:` 分隔，例如，`?,224,224,3:?,300,300,1`（默认值：）
- output_nodes：字符串：要量化的子计算图的输出节点名称列表（以逗号分隔），搭配 input_nodes 一起使用。生成部署模型时，仅包含 input_nodes 与 output_nodes 之间的子计算图。请将其设置为要量化的模型主体结束位置，例如，数据后处理之前的节点。（默认值：）
- input_fn：字符串：用于提供输入数据的 Python 可导入函数。格式为 `module_name.input_fn_name`，例如，`my_input_fn.input_fn`。`input_fn` 应采用 `int` 对象作为输入，以指示校准步骤，并给每次调用返回 dict (`placeholder_node_name : numpy.Array`) 对象，此对象将馈送到模型的占位符节点中。（默认值：）
- method：int32：{0,1,2}，默认值：1。量化方法，选项包括：
  - 0：非上溢方法。确保量化期间没有任何值达到饱和。它可能导致结果不准确

    - 1：最小差值方法。支持在量化期间较大值达到饱和，以获得较小的量化误差。它比方法 0 慢，但对离群值有更高的耐受力。
    - 2：使用逐通道策略的最小差值方法。支持在量化期间较大值达到饱和，以获得较小的量化误差。为逐通道权重应用特殊策略，但对标准权重和激活则实现方法 1。它比方法 0 慢，但对离群值有更高的耐受力。
- calib_iter：int32。校准的迭代。校准图像总数 = calib_iter * batch_size（默认值：100）
- output_dir：字符串。用于保存量化结果的目录（默认：./quantize_results）。

**注释：**如需了解有关 `**kargs` 的其他参数的更多信息，请参阅 vai_q_tensorflow 用法。

**注释：**如需了解有关 WeGO TensorFlow 1.x 的即时量化的更多信息，请参阅示例。

# 使用 AMD ZenDNN 优化性能

ZenDNN 库包含多个 API 用于专为 AMD CPU 架构而最优化的基本神经网络构建块，此库专供深度学习应用和框架开发者用于改善 AMD CPU 上的深度学习推断性能。为了改善 DPU 不支持的运算符、CPU（尤其是 AMD CPU）的性能，ZenDNN 现已集成到 WeGO 流程中。

**注释：**这一项实验性功能。当前不保证在 WeGO 中使用 ZenDNN 所能获得的性能增益。您可根据子集的需求启用或禁用 ZenDNN。

## WeGO PyTorch 中的 ZenDNN

### 在 WeGO PyTorch 中启用 ZenDNN

ZenDNN 初始为禁用状态，但您可选择通过 WeGO-Torch 的编译 API 将其启用：

```
wego_mod = wego_torch.compile(mod, wego_torch.CompileOptions(
...
optimize_options = wego_torch.OptimizeOptions(zendnn_enable = True))
)
```

启用 ZenDNN 后，已编译的 WeGO 计算图中的 CPU 运算符（DPU 不支持的运算符）将被替换为 ZenDNN 运算符，并且会使用 ZenDNN 内核执行这些运算符以供加速。

### 环境变量

ZenDNN 提供了部分环境变量用于性能调优。

表 51：**环境变量**

| 名称 | 描述 |
|---|---|
| OMP_DYNAMIC | 要启用 ZenDNN 时，请将其显式设置为 FALSE。 |
| ZENDNN_GEMM_ALGO | 默认值为 3。您可设置 [0, 1, 2, 3, 4] 来调整不同 GEMM ALGO 路径。 |
| OMP_NUM_THREADS | 默认值是用户系统的物理核数。您需要根据推断线程数进行调整以达成更好的性能。欲知详情，请参阅调谐准则。 |

**调谐准则**

ZenDNN 使用 OpenMP 作为底层库。OMP_NUM_THREADS 环境变量用于控制运算符内部并行度，即 ZenDNN 内核中的多核并行度。对于 OpenMP，不同的应用线程或运算符间的线程可使用不同的 OpenMP 线程池来处理运算符内的任务。因此，在多线程应用程序中可能会使用许多 OpenMP 线程，这将耗用大量 CPU 内核资源并降低整体性能。因此，建议的 OMP_NUM_THREADS 调谐值是根据目标 CPU 平台中的核心数以及应用中使用的线程数设置的，以避免超额预订。例如，如果您在单个应用中启动 16 个线程，并且您的平台上有 64 个 CPU 核，那么您可设置 OMP_NUM_THREADS <= 4 以避免 CPU 核争用。

## WeGO TensorFlow 2 中的 ZenDNN

### 在 WeGO TensorFlow 2 中启用 ZenDNN

默认禁用 ZenDNN。设置 export TF_ENABLE_ZENDNN_OPTS=1 以将其启用。

**环境变量**

您必须显式导出以下环境变量才能在 WeGO TensorFlow 2 中启用并正确运行 ZenDNN。

表 52：**环境变量**

| 名称 | 描述 |
|---|---|
| OMP_DYNAMIC | 启用 ZenDNN 时，请将该变量显式设为 FALSE。 |
| OMP_NUM_THREADS | 显式设置该变量以达成更好的性能。请参阅调谐准则以获取更多详细信息。 |
| ZENDNN_GEMM_ALGO | 默认值为 3。您可设置 [0, 1, 2, 3, 4] 来调整不同 GEMM ALGO 路径。 |
| ZENDNN_TENSOR_POOL_LIMIT | 默认值为 32。请参阅调谐准则以获取更多详细信息。 |
| ZENDNN_TENSOR_BUF_MAXSIZE_ENABLE | 默认值为 0。<br>・ 0：启用减小存储器池张量。<br>・ 1：启用增大存储器池张量。 |
| TF_ENABLE_ZENDNN_OPTS | 默认值为 0。设为 1 启用 ZenDNN。 |

**调谐准则**

根据用户系统的核数设置 OMP_NUM_THREADS。AMD 建议设置较小的数值，如 1 或 2。

在某些情况下，请将 ZENDNN_TENSOR_POOL_LIMIT 设为较小的数值，如 1，这样某些层会在张量池达到 ZENDNN_TENSOR_POOL_LIMIT 的池限值时，使用默认存储器分配代替张量池。

第 6 章

# 模型剖析

## Vitis AI Profiler

AMD Vitis™ AI Profiler 是一组工具，可帮助基于 VART 对 AI 应用进行剖析和可视化：

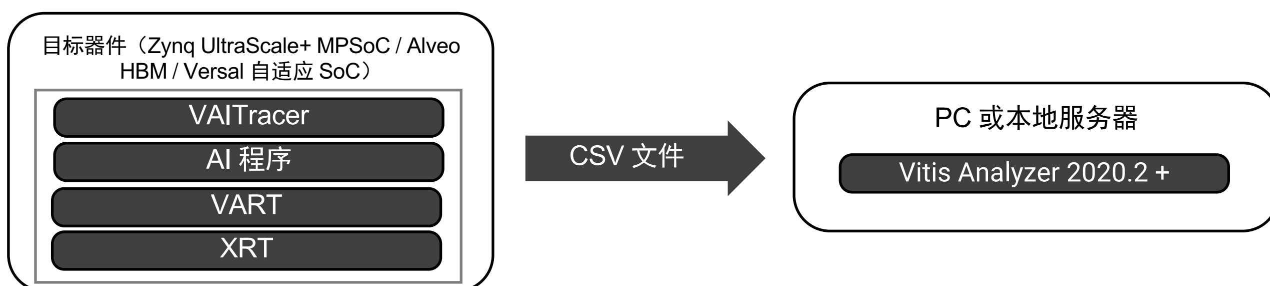
- 它无需对用户代码进行任何更改，也无需对程序进行任何再编译，因此非常便于使用。
- 直观显示系统性能瓶颈。
- 展示不同计算单元 (CPU/DPU) 的执行状态。

Vitis AI Profiler 是 Vitis AI 全功能一体化剖析解决方案。它属于应用级工具，可基于 VART 对 AI 应用进行剖析和可视化。对于 AI 应用来说，有些组件是在硬件上运行的。例如，神经网络计算通常在 DPU 上运行，有些组件则作为 C/C++ 代码实现的函数在 CPU 上运行（如图像预处理）。该工具可帮助您将所有这些不同组件的运行状态整合在一起。

### Vitis AI Profiler 架构

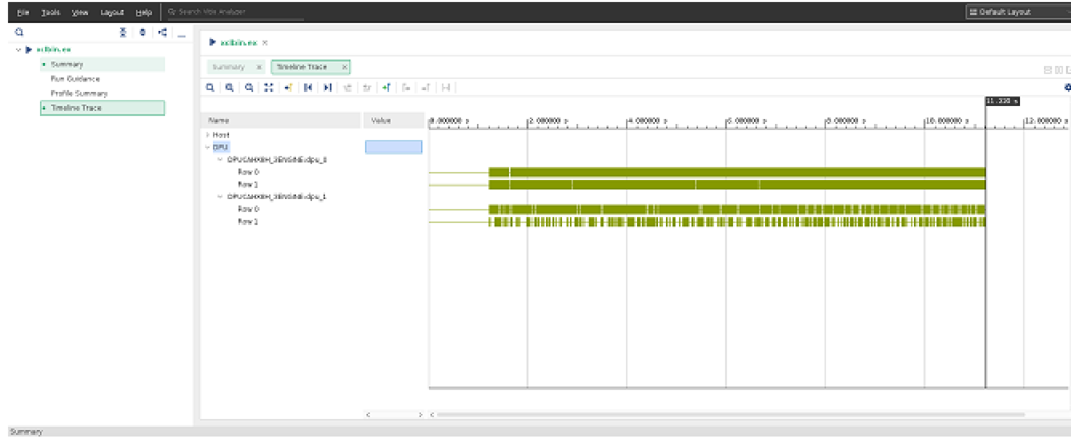
Vitis AI Profiler 架构如下图所示：

图 32：**Vitis AI Profiler 架构**



# Vitis AI Profiler GUI 概述

图 33：**Vitis AI Profiler GUI 概述**



- DPU 摘要：有关每个内核的运行轮数和最短、平均、最长时间 (ms) 的汇总表。

图 34：**DPU 摘要**

xclbin.ex ×

Summary × | Profile Summary × | Timeline Trace ×

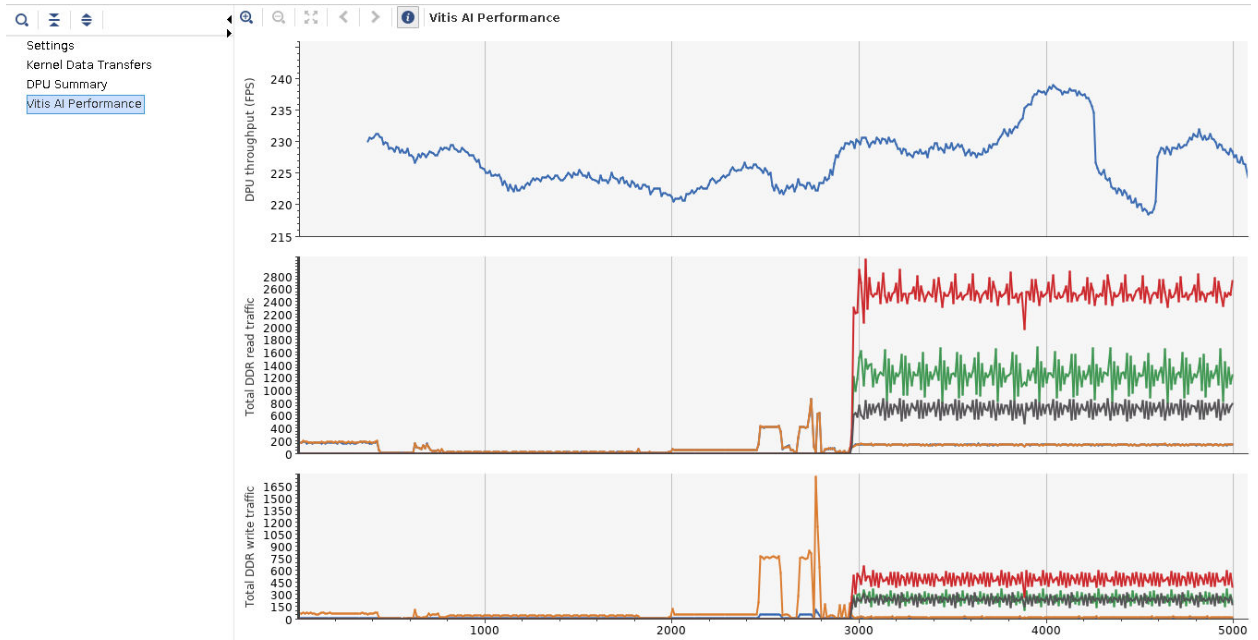
Settings
DPU Summary
Vitis AI Performance

**DPU Summary**

| Kernel | Compute Unit | Runs | Min Time (ms) | Avg Time (ms) | Max Time (ms) | Workload (GOP) | Performance (GOP/s) | Mem IO (MB) | Mem Bandwidth (MB/s) |
|---|---|---|---|---|---|---|---|---|---|
| subgraph_res2b | DPUCZDX8G_1:batch-1 | 181 | 0.514 | 0.519 | 0.585 | 0.103 | 197.997 | 1.823 | 3,512.478 |
| subgraph_res2c_branch2a | DPUCZDX8G_1:batch-1 | 181 | 0.250 | 0.255 | 0.412 | 0.103 | 402.32 | 1.02 | 3,993.299 |
| subgraph_res2c_branch2b | DPUCZDX8G_1:batch-1 | 181 | 0.345 | 0.350 | 0.363 | 0.231 | 661.187 | 0.438 | 1,253.497 |
| subgraph_res2c | DPUCZDX8G_1:batch-1 | 181 | 0.513 | 0.517 | 0.528 | 0.103 | 198.876 | 1.823 | 3,528.064 |
| subgraph_fake_downsample_3 | DPUCZDX8G_1:batch-1 | 181 | 0.163 | 0.168 | 0.178 | 0 | 0 | 0.401 | 2,396.426 |
| subgraph_fake_downsample_0 | DPUCZDX8G_1:batch-1 | 181 | 0.166 | 0.170 | 0.189 | 0 | 0 | 0.401 | 2,359.537 |
| subgraph_res3a_branch2a | DPUCZDX8G_1:batch-1 | 181 | 0.160 | 0.164 | 0.185 | 0.051 | 313.157 | 0.334 | 2,035.401 |
| subgraph_res3a_branch2b | DPUCZDX8G_1:batch-1 | 181 | 0.327 | 0.331 | 0.342 | 0.231 | 698.092 | 0.348 | 1,051.58 |
| subgraph_res3a_branch2c | DPUCZDX8G_1:batch-1 | 181 | 0.274 | 0.278 | 0.290 | 0.103 | 369.826 | 0.568 | 2,043.49 |
| subgraph_res3a | DPUCZDX8G_1:batch-1 | 181 | 0.553 | 0.558 | 0.568 | 0.206 | 368.386 | 1.135 | 2,034.619 |
| subgraph_res3b_branch2a | DPUCZDX8G_1:batch-1 | 181 | 0.213 | 0.216 | 0.227 | 0.103 | 474.868 | 0.567 | 2,622.134 |
| subgraph_res3b_branch2b | DPUCZDX8G_1:batch-1 | 181 | 0.335 | 0.340 | 0.485 | 0.231 | 680.077 | 0.348 | 1,024.443 |
| subgraph_res3b | DPUCZDX8G_1:batch-1 | 181 | 0.332 | 0.336 | 0.348 | 0.103 | 305.744 | 0.969 | 2,883.718 |
| subgraph_res3c_branch2a | DPUCZDX8G_1:batch-1 | 181 | 0.213 | 0.216 | 0.227 | 0.103 | 474.977 | 0.567 | 2,622.737 |
| subgraph_res3c_branch2b | DPUCZDX8G_1:batch-1 | 181 | 0.335 | 0.339 | 0.350 | 0.231 | 681.572 | 0.348 | 1,026.695 |
| subgraph_res3c | DPUCZDX8G_1:batch-1 | 181 | 0.331 | 0.336 | 0.345 | 0.103 | 305.986 | 0.969 | 2,885.995 |
| subgraph_res3d_branch2a | DPUCZDX8G_1:batch-1 | 181 | 0.212 | 0.216 | 0.226 | 0.103 | 474.868 | 0.567 | 2,622.134 |
| subgraph_res3d_branch2b | DPUCZDX8G_1:batch-1 | 181 | 0.335 | 0.340 | 0.509 | 0.231 | 679.282 | 0.348 | 1,023.246 |
| subgraph_res3d | DPUCZDX8G_1:batch-1 | 181 | 0.332 | 0.336 | 0.348 | 0.103 | 305.573 | 0.969 | 2,882.107 |
| subgraph_fake_downsample_4 | DPUCZDX8G_1:batch-1 | 181 | 0.113 | 0.117 | 0.127 | 0 | 0 | 0.201 | 1,715.095 |
| subgraph_fake_downsample_1 | DPUCZDX8G_1:batch-1 | 181 | 0.113 | 0.117 | 0.131 | 0 | 0 | 0.201 | 1,715.662 |
| subgraph_res4a_branch2a | DPUCZDX8G_1:batch-1 | 181 | 0.145 | 0.149 | 0.167 | 0.051 | 344.349 | 0.282 | 1,888.989 |
| subgraph_res4a_branch2b | DPUCZDX8G_1:batch-1 | 181 | 0.318 | 0.322 | 0.339 | 0.231 | 717.53 | 0.69 | 2,142.655 |
| subgraph_res4a_branch2c | DPUCZDX8G_1:batch-1 | 181 | 0.298 | 0.303 | 0.320 | 0.103 | 339.385 | 0.514 | 1,697.735 |
| subgraph_res4a | DPUCZDX8G_1:batch-1 | 181 | 0.529 | 0.534 | 0.556 | 0.206 | 384.652 | 1.027 | 1,922.262 |
| subgraph_res4b_branch2a | DPUCZDX8G_1:batch-1 | 181 | 0.210 | 0.215 | 0.245 | 0.103 | 478.534 | 0.513 | 2,390.236 |
| subgraph_res4b_branch2b | DPUCZDX8G_1:batch-1 | 181 | 0.318 | 0.323 | 0.345 | 0.231 | 715.909 | 0.69 | 2,137.816 |
| subgraph_res4b | DPUCZDX8G_1:batch-1 | 181 | 0.439 | 0.445 | 0.465 | 0.103 | 230.756 | 0.715 | 1,605.028 |
| subgraph_res4c_branch2a | DPUCZDX8G_1:batch-1 | 181 | 0.209 | 0.214 | 0.225 | 0.103 | 479.706 | 0.513 | 2,396.092 |
| subgraph_res4c_branch2b | DPUCZDX8G_1:batch-1 | 181 | 0.317 | 0.323 | 0.334 | 0.231 | 716.289 | 0.69 | 2,138.951 |
| subgraph_res4c | DPUCZDX8G_1:batch-1 | 181 | 0.440 | 0.445 | 0.477 | 0.103 | 230.785 | 0.715 | 1,605.228 |

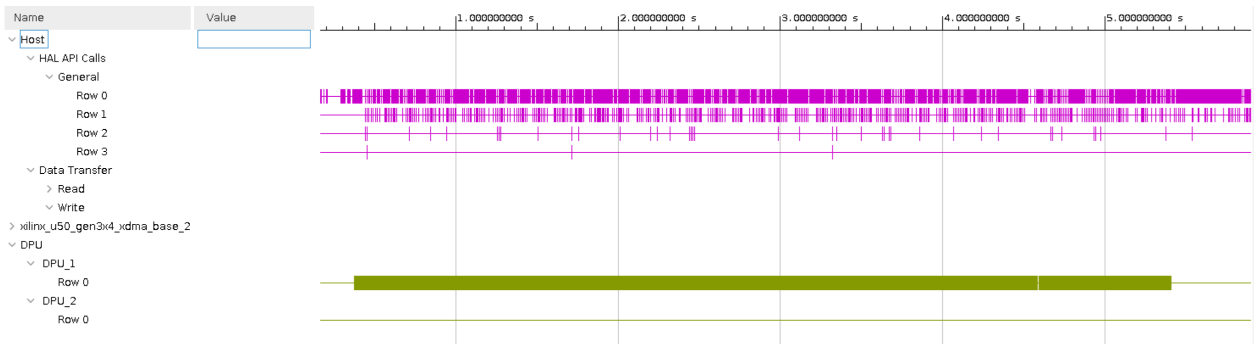
- DPU 吞吐量和 DDR 传输速率：应用运行期间实现的 FPS 和采样所得的读写传输速率 (Mbps) 的折线图。

图 35：**DPU 吞吐量和 DDR 传输速率**




- 时间线轨迹：包含来自 VART、HAL API 和 DPU 的计时事件。

图 36：**时间线轨迹**




**注释：**Vitis Analyzer 是默认 GUI，用于 Vitis AI 1.3 和更高版本的 vaitrace。

# Vitis AI Profiler 入门指南

### 系统要求

- 硬件：
  - 支持 AMD Zynq™ UltraScale+™ MPSoC (DPUCZDX8G)
    
    支持 AMD Versal™ 自适应 SoC (DPUCVDX8G/ DPUCVDX8H)
- 软件：
  - 支持 VART v1.2+

## 安装 Vitis AI Profiler

1. 在 Zynq UltraScale+ MPSoC PetaLinux 平台中为 vaitrace 准备调试环境。
    a. 运行 `petalinux-config -c kernel` 配置和构建 PetaLinux。
    b. 为 Linux 内核启用以下设置。
        - General architecture-dependent options ---> [*] Kprobes
        - Kernel hacking ---> [*] Tracers
        - Kernel hacking ---> [*] Tracers --->

          [*] Kernel Function Tracer

          [*] Enable kprobes-based dynamic events

          [*] Enable uprobes-based dynamic events
    c. 运行 `petalinux-config -c rootfs` 并为 root-fs 启用以下设置。

       Petalinux package Groups ---> packaggroup-petalinux-self-hosted ---> [*] packagegroup-petalinux-self-hosted
    d. 运行 `petalinux-build`。
2. 安装 vaitrace。vaitrace 已集成到 VART 运行时中。如果已安装 VART 运行时，vaitrace 会安装到 `/usr/bin/vaitrace` 中。

## 使用 vaitrace 启动简单追踪

以下示例使用的是 VART ResNet50 样本：

1. 下载并设置 Vitis AI。
2. 启动测试并追踪。
    - 对于 C++ 程序，在测试命令前添加 `vaitrace`，如下所示：

```
# cd ~/Vitis_AI/examples/vai_runtime/resnet50
# vaitrace ./resnet50 /usr/share/vitis_ai_library/models/resnet50/
resnet50.xmodel
```

    - 对于 Python 程序，在 Python 解释器命令后添加 `-m vaitrace_py`，如下所示：

```
# cd ~/Vitis_AI/examples/vai_runtime/resnet50_mt_py
# python3 -m vaitrace_py ./resnet50.py 2 /usr/share/vitis_ai_library/
models/resnet50/resnet50.xmodel
```

   vaitrace 和 XRT 会在工作目录中生成某些文件。
3. 请将所有 .csv 文件和 `xclbin.ex.run_summary` 都复制到您的系统。您可使用 vitis_analyzer 2020.2 和更高版本来打开 `xclbin.ex.run_summary`：
    - 如果使用命令行，请运行 `# vitis_analyzer xclbin.ex.run_summary`。
    - 如果使用 GUI，请选择“File” → “Open Summary” → “xclbin.ex.run_summary” （文件 > 打开汇总 > xclbin.ex.run_summary）。

如需了解有关 Vitis 分析器的更多信息，请参阅《Vitis 统一软件平台文档：应用加速开发》(UG1393) 中的使用“Analysis”视图 (Vitis Analyzer)。

# vaitrace 用法

**命令行用法**

```
# vaitrace --help
usage: vaitrace [-h] [-c [CONFIG]] [-d] [-o [TRACESAVETO]] [-t [TIMEOUT]] [-
v]
                [-b] [-p] [--va] [--xat] [--txt_summary] [--fine_grained]
                ...

positional arguments:
  cmd

optional arguments:
  -h, --help          show this help message and exit
  -c [CONFIG]         Specify the config file
  -d                  Enable debug
  -o [TRACESAVETO]    Save report to, only available for txt summary mode
  -t [TIMEOUT]        Tracing time limitation
  -v                  Show version
  -b                  Bypass vaitrace, just run command
  -p                  Trace python application
  --va                Generate trace data for Vitis Analyzer
  --xat               Save raw data for debug usage
  --txt_summary       Display txt summary
  --fine_grained      Fine-grained mode
```

以下提供了一些重要且常用的实参：

- cmd：cmd 是一个可执行的 Vitis AI 追踪程序，包括程序名称和参数。
- -t：控制追踪时间（以毫秒为单位），从启动 [cmd] 开始计算，默认值为 30。换言之，如果，针对 vaitrace 不指定 -t，那么在 [cmd] 持续运行 30 秒后停止追踪。[cmd] 会继续正常运行，但停止收集追踪数据。
- -c：您可采用更多定制选项来启动追踪，只需在 JSON 配置文件上写入这些选项，并使用 -c 指定配置即可。配置文件详细信息会下一章节中详细解释。
- -o：报告的位置。此实参仅在文本汇总模式下可用。测试汇总信息默认输出到 STDOUT。
- --va：为 Vitis 分析器生成追踪数据，默认为启用，不可与 --txt_summary 搭配使用。
- --txt_summary 或 --txt：输出文本汇总。在此模式下，vaitrace 不会为 Vitis 分析器生成报告，不可与 --va 搭配使用。
- --fine_grained：在高精度模式下启动追踪。此模式会生成大量追踪数据，追踪时间以 10 秒为限。

其他实参用于调试。

# 配置

建议使用配置文件来记录 vaitrace 的追踪选项。您可使用 `vaitrace -c trace_cfg.json` 通过配置启动追踪。

配置优先级：“Configuration File” → “Command Line” → “Default” （配置文件 > 命令行 > 默认）。

以下提供了一个 vaitrace 配置文件示例。

```
{
  "trace": {
      "enable_trace_list": ["vitis-ai-library", "vart", "custom"]
  }
  "trace_custom": []
}
```

表 53：**配置文件内容**

| 密钥名称 | | 值类型 | 描述 |
|---|---|---|---|
| trace | | 对象 | |
| | enable_trace_list | list | 要启用的内置追踪函数列表，可用值包括：vitis-ai-library、vart、opencv 和 custom。对于 trace_custom 列表中的函数，该值可定制。 |
| trace_custom | | list | 要追踪的函数列表，这些函数由用户来实现。对于函数名称，支持命名空间。您可在本文档后文中看到使用定制追踪函数的示例。 |

## 文本汇总

使用 `--txt` 或 `--txt_summary` 选项时，vaitrace 将打印 ASCII 表，如下图所示：

图 37：**ASCII 表**

DPU Summary:

| DPU Id | Bat | SubGraph | WL | RT | Perf | LdWB | LdFM | StFM | AvgBw |
|---|---|---|---|---|---|---|---|---|---|
| DPUCZDX8G_1 | 1 | conv1 | 0.239 | 0.612 | 389.930 | 0.009 | 0.507 | 0.191 | 1183.568 |
| DPUCZDX8G_1 | 1 | res2a_branch2a | 0.026 | 0.189 | 136.988 | 0.004 | 0.191 | 0.191 | 2095.569 |
| DPUCZDX8G_1 | 1 | res2a_branch2b | 0.231 | 0.285 | 811.971 | 0.035 | 0.191 | 0.191 | 1501.974 |
| DPUCZDX8G_1 | 1 | res2a_branch2c | 0.104 | 0.270 | 383.568 | 0.016 | 0.191 | 0.766 | 3689.815 |
| DPUCZDX8G_1 | 1 | res2a | 0.105 | 0.566 | 185.811 | 0.016 | 0.957 | 0.766 | 3145.318 |
| DPUCZDX8G_1 | 1 | res2b_branch2a | 0.103 | 0.242 | 425.459 | 0.016 | 0.766 | 0.191 | 4115.961 |
| DPUCZDX8G_1 | 1 | res2b_branch2b | 0.231 | 0.334 | 692.849 | 0.035 | 0.191 | 0.191 | 1281.624 |
| DPUCZDX8G_1 | 1 | res2b | 0.105 | 0.491 | 214.193 | 0.016 | 0.957 | 0.766 | 3625.764 |
| DPUCZDX8G_1 | 1 | res2c_branch2a | 0.103 | 0.243 | 423.708 | 0.016 | 0.766 | 0.191 | 4099.023 |
| DPUCZDX8G_1 | 1 | res2c_branch2b_bias | 0.058 | 0.163 | 354.926 | 0.035 | 0.191 | 0.048 | 1724.310 |
| DPUCZDX8G_1 | 1 | res2b_downsample_bp_by_fake_downsample_0 | 0.026 | 0.270 | 97.379 | 0.016 | 0.239 | 0.191 | 1693.519 |
| DPUCZDX8G_1 | 1 | res3a_branch2a | 0.051 | 0.167 | 308.267 | 0.031 | 0.191 | 0.096 | 1952.844 |
| DPUCZDX8G_1 | 1 | res3a_branch2b | 0.231 | 0.323 | 716.134 | 0.141 | 0.096 | 0.096 | 1053.019 |
| DPUCZDX8G_1 | 1 | res3a_branch2c | 0.103 | 0.266 | 387.827 | 0.063 | 0.096 | 0.383 | 2084.586 |
| DPUCZDX8G_1 | 1 | res3a | 0.207 | 0.527 | 392.268 | 0.125 | 0.574 | 0.383 | 2103.416 |
| DPUCZDX8G_1 | 1 | res3b_branch2a | 0.103 | 0.209 | 492.157 | 0.063 | 0.383 | 0.096 | 2651.316 |
| DPUCZDX8G_1 | 1 | res3b_branch2b | 0.231 | 0.323 | 716.134 | 0.141 | 0.096 | 0.096 | 1053.019 |
| DPUCZDX8G_1 | 1 | res3b | 0.104 | 0.319 | 325.908 | 0.063 | 0.479 | 0.383 | 2967.085 |
| DPUCZDX8G_1 | 1 | res3c_branch2a | 0.103 | 0.209 | 492.157 | 0.063 | 0.383 | 0.096 | 2651.316 |
| DPUCZDX8G_1 | 1 | res3c_branch2b | 0.231 | 0.323 | 716.134 | 0.141 | 0.096 | 0.096 | 1053.019 |
| DPUCZDX8G_1 | 1 | res3c | 0.104 | 0.320 | 324.890 | 0.063 | 0.479 | 0.383 | 2957.813 |
| DPUCZDX8G_1 | 1 | res3d_branch2a | 0.103 | 0.208 | 494.523 | 0.063 | 0.383 | 0.096 | 2664.063 |
| DPUCZDX8G_1 | 1 | res3d_branch2b_bias | 0.058 | 0.142 | 407.238 | 0.141 | 0.096 | 0.024 | 1877.641 |
| DPUCZDX8G_1 | 1 | res3c_downsample_bp_by_fake_downsample_1 | 0.026 | 0.209 | 124.360 | 0.063 | 0.120 | 0.096 | 1363.636 |
| DPUCZDX8G_1 | 1 | res4a_branch2a | 0.051 | 0.144 | 357.156 | 0.125 | 0.096 | 0.048 | 1911.458 |
| DPUCZDX8G_1 | 1 | res4a_branch2b | 0.231 | 0.307 | 753.294 | 0.563 | 0.048 | 0.048 | 2196.254 |
| DPUCZDX8G_1 | 1 | res4a_branch2c | 0.103 | 0.289 | 356.267 | 0.251 | 0.048 | 0.191 | 1737.024 |
| DPUCZDX8G_1 | 1 | res4a | 0.206 | 0.504 | 408.974 | 0.501 | 0.287 | 0.191 | 1990.079 |
| DPUCZDX8G_1 | 1 | res4b_branch2a | 0.103 | 0.206 | 499.081 | 0.250 | 0.191 | 0.048 | 2433.252 |
| DPUCZDX8G_1 | 1 | res4b_branch2b | 0.231 | 0.308 | 750.848 | 0.563 | 0.048 | 0.048 | 2189.123 |
| DPUCZDX8G_1 | 1 | res4b | 0.103 | 0.423 | 244.356 | 0.251 | 0.239 | 0.191 | 1650.118 |
| DPUCZDX8G_1 | 1 | res4c_branch2a | 0.103 | 0.206 | 499.081 | 0.250 | 0.191 | 0.048 | 2433.252 |
| DPUCZDX8G_1 | 1 | res4c_branch2b | 0.231 | 0.307 | 753.294 | 0.563 | 0.048 | 0.048 | 2196.254 |
| DPUCZDX8G_1 | 1 | res4c | 0.103 | 0.420 | 246.101 | 0.251 | 0.239 | 0.191 | 1661.905 |
| DPUCZDX8G_1 | 1 | res4d_branch2a | 0.103 | 0.205 | 501.515 | 0.250 | 0.191 | 0.048 | 2445.122 |
| DPUCZDX8G_1 | 1 | res4d_branch2b | 0.231 | 0.306 | 755.756 | 0.563 | 0.048 | 0.048 | 2203.431 |
| DPUCZDX8G_1 | 1 | res4d | 0.103 | 0.422 | 244.935 | 0.251 | 0.239 | 0.191 | 1654.028 |
| DPUCZDX8G_1 | 1 | res4e_branch2a | 0.103 | 0.205 | 501.515 | 0.250 | 0.191 | 0.048 | 2445.122 |
| DPUCZDX8G_1 | 1 | res4e_branch2b | 0.231 | 0.307 | 753.294 | 0.563 | 0.048 | 0.048 | 2196.254 |
| DPUCZDX8G_1 | 1 | res4e | 0.103 | 0.421 | 245.517 | 0.251 | 0.239 | 0.191 | 1657.957 |
| DPUCZDX8G_1 | 1 | res4f_branch2a | 0.103 | 0.206 | 499.081 | 0.250 | 0.191 | 0.048 | 2433.252 |
| DPUCZDX8G_1 | 1 | res4f_branch2b_bias | 0.058 | 0.159 | 363.618 | 0.563 | 0.048 | 0.012 | 4009.434 |
| DPUCZDX8G_1 | 1 | res4e_downsample_bp_by_fake_downsample_2 | 0.026 | 0.248 | 104.196 | 0.251 | 0.203 | 0.048 | 2073.589 |
| DPUCZDX8G_1 | 1 | res5a_branch2a | 0.051 | 0.160 | 321.283 | 0.500 | 0.048 | 0.024 | 3662.500 |
| DPUCZDX8G_1 | 1 | res5a_branch2b | 0.231 | 0.391 | 591.397 | 2.250 | 0.024 | 0.024 | 6019.182 |
| DPUCZDX8G_1 | 1 | res5a_branch2c | 0.103 | 0.341 | 301.645 | 1.002 | 0.024 | 0.096 | 3368.035 |
| DPUCZDX8G_1 | 1 | res5a | 0.206 | 0.438 | 469.913 | 2.002 | 0.144 | 0.096 | 5239.726 |
| DPUCZDX8G_1 | 1 | res5b_branch2a | 0.103 | 0.234 | 439.254 | 1.000 | 0.096 | 0.024 | 4901.709 |
| DPUCZDX8G_1 | 1 | res5b_branch2b | 0.231 | 0.394 | 586.894 | 2.250 | 0.024 | 0.024 | 5973.350 |
| DPUCZDX8G_1 | 1 | res5b | 0.103 | 0.377 | 273.373 | 1.002 | 0.120 | 0.096 | 3305.366 |
| DPUCZDX8G_1 | 1 | res5c_branch2a | 0.103 | 0.232 | 443.041 | 1.000 | 0.096 | 0.024 | 4943.966 |
| DPUCZDX8G_1 | 1 | res5c_branch2b | 0.231 | 0.391 | 591.397 | 2.250 | 0.024 | 0.024 | 6019.182 |
| DPUCZDX8G_1 | 1 | res5c | 0.103 | 0.374 | 275.566 | 1.002 | 0.120 | 0.096 | 3332.888 |
| DPUCZDX8G_1 | 1 | pool5 | ~0 | 0.102 | 0.084 | ~0 | 0.096 | 0.002 | 980.392 |
| DPUCZDX8G_1 | 1 | fc1000_bias | 0.004 | 0.312 | 13.131 | 1.954 | 0.002 | ~0 | 6422.927 |

```
Notes:
"~0": Within range of (0, 0.001)
Bat: DPU batch number
WL(GOP): Workload
RT(ms): Run time
Perf(GOP/s)
LdFM(MB): External memory load size of feature map
LdBW(MB): External memory load size of bias and weight
StFM(MB): External memory store size of feature map
AvgBw(MB/s): External memory aaverage bandwidth
....
```

CPU Tasks in Graph(called by graph runner):

| SubGraph | OPs | Device | Runs | AverageRunTime(ms) |
|---|---|---|---|---|
| fc1000_fixed_ | fix2float | CPU | 1 | 0.063 |

CPU Functions(Not in Graph, e.g.: pre/post-processing, vai-runtime):

| Function | Device | Runs | AverageRunTime(ms) |
|---|---|---|---|
| xir::XrtCu::run | CPU | 55 | 0.295 |

以下列表中定义了这些字段：

- DPU Id：DPU 实例的名称。
- Bat：DPU 实例的批次大小。
- SubGraph：XMODEL 中的子计算图名称。
- WL (Workload)：计算工作负载（MAC 指示 2 次运算）。单位为 GOP。
- RT (Runtime)：以毫秒表示的执行时间，单位是 ms。
- Perf：DPU 性能（以每秒 GOP 数为单位）。单位为 GOP/秒。
- LdFM (Load Size of Feature Map)：特征映射的外部存储器加载大小。单位为 MB。

- LdWB (Load Size of Weight and Bias)：偏差和权重的外部存储器负载大小。单位为 MB。
- StFM (Store Size of Feature Map)：特征映射的外部存储器存储大小。单位为 MB。
- AvgBw (Average bandwidth)：DDR 存储器访问平均带宽。

  AvgBw =（子计算图的总负载大小（包括特征映射和权重/偏差，从 DDR/HBM 到 DPU bank mem）+ 子计算图的总存储大小（从 DPU bank mem 到 DDR/HBM））/ 子计算图运行时间

## DPU 剖析示例

您可在 Vitis AI Profiler GitHub 页面上找到使用 Vitis AI Profiler 的高级 DPU 剖析示例。

附录 A

# 错误代码

表 54：**错误代码**

| 错误代码 ID | 错误消息 |
|---|---|
| OPTIMIZER_DATA_PARALLEL_NOT_ALLOWED_ERROR | torch.nn.DataParallel module is not allowed. |
| OPTIMIZER_INVALID_ANA_RESULT_ERROR | Model analysis result is not valid. This is usually caused by PyTorch or Python version changes. |
| OPTIMIZER_INVALID_ARGUMENT_ERROR | Invalid argument. |
| OPTIMIZER_TORCH_MODULE_ERROR | The operation is not an instance of torch.nn.Module. |
| OPTIMIZER_NOT_EXCLUDE_NODE_ERROR | Some nodes must be excluded from pruning. |
| OPTIMIZER_NO_ANA_RESULT_ERROR | Model analysis results not found. |
| OPTIMIZER_SUBNET_ERROR | Subnet candidates not found. Must do subnet searching first. |
| OPTIMIZER_UNSUPPORTED_OP_ERROR | The operation is not supported yet. |
| OPTIMIZER_KERAS_MODEL_ERROR | The given object is not an instance of keras.Model. |
| OPTIMIZER_KERAS_LAYER_ERROR | The operation is not an instance of keras.Layer. |
| OPTIMIZER_DATA_FORMAT_ERROR | The data format for saving weights is not allowed in pruning. |
| OPTIMIZER_INVALID_GRAPH | The parsed graph is invalid. |
| OPTIMIZER_IO_ERROR | IO error. Usually occurs during disk read/write. |
| OPTIMIZER_MODEL_ANALYSIS_ERROR | An error occurred while performing model analysis. |
| OPTIMIZER_PARSE_GRAPH_FAILED | Unable to parse the model to a computation graph. |
| OPTIMIZER_WEIGHTS_NOT_FOUND | The weights for the operation can not be found. |
| QUANTIZER_TF1_INVALID_BITWIDTH | invalid parameter |
| QUANTIZER_TF1_INVALID_METHOD | invalid parameter |
| QUANTIZER_TF1_INVALID_TARGET_DTYPE | invalid parameter |
| QUANTIZER_TF1_MISSING_QUANTIZE_INFO | not found |
| QUANTIZER_TF1_INVALID_INPUT | not found |
| QUANTIZER_TF1_UNSUPPORTED_OP | Unsupported Op type |
| QUANTIZER_TF1_LENGTH_MISMATCH | invalid parameter |
| QUANTIZER_TF1_INVALID_INPUT_FN | fail to import |
| QUANTIZER_TF2_UNSUPPORTED_MODEL | Unsupported model type |
| QUANTIZER_TF2_UNSUPPORTED_LAYER | Unsupported layer type |
| QUANTIZER_TF2_INVALID_CALIB_DATASET | Invalid calibration dataset |
| QUANTIZER_TF2_INVALID_INPUT_SHAPE | Invalid input shape |
| QUANTIZER_TF2_INVALID_TARGET | Invalid Target |
| QUANTIZER_TORCH_BIAS_CORRECTION | Bias correction file in quantization result directory does not match current model. |

表 54：**错误代码** (续)

| 错误代码 ID | 错误消息 |
|---|---|
| QUANTIZER_TORCH_CALIB_RESULT_MISMATCH | Node name mismatch is found when loading quantization steps of tensors. Please ensure the vai_q_pytorch version and PyTorch version for test mode are the same as those in calibration (or QAT training) mode. |
| QUANTIZER_TORCH_EXPORT_ONNX | The quantized module, which is based PyTorch traced model, can not be exported to ONNX due to PyTorch internal failure. The PyTorch internal failure reason is listed in message text. May needs adjust the float model code. |
| QUANTIZER_TORCH_EXPORT_XMODEL | Fail to convert the graph to XMODEL. Needs to check the reasons in the message text. |
| QUANTIZER_TORCH_FAST_FINETINE | Fast fine-tuned parameter file does not exist. Call load_ft_paramthe in the model code to load them. |
| QUANTIZER_TORCH_FIX_INPUT_TYPE | Data type or value is illegal in arguments of quantization OP when exporting the ONNX model. |
| QUANTIZER_TORCH_ILLEGAL_BITWIDTH | The configuration of tensor quantization is illegal. It should be an integer and in the range given in message text. |
| QUANTIZER_TORCH_IMPORT_KERNEL | Importing vai_q_pytorch library file error. Check PyTorch version matching vai_q_pytorch version (pytorch_nndct.__version__) or not. |
| QUANTIZER_TORCH_NO_CALIB_RESULT | Quantization result file does not exist. Check whether the calibration is done or not. |
| QUANTIZER_TORCH_NO_CALIBRATION | Quantization calibration is not performed completely, check if module FORWARD function is called! FORWARD function of torch_quantizer.quant_model needs to be called in the user code explicitly. Please refer to the example code at https://github.com/Xilinx/Vitis-AI/blob/master/src/Vitis-AI-Quantizer/vai_q_pytorch/example/resnet18_quant.py. |
| QUANTIZER_TORCH_NO_FORWARD | torch_quantizer.quant_model FORWARD function must be called before exporting quantization result. Please refer to example code at https://github.com/Xilinx/Vitis-AI/blob/master/src/Vitis-AI-Quantizer/vai_q_pytorch/example/resnet18_quant.py. |
| QUANTIZER_TORCH_OP_REGIST | The type of OP can't be registered multiple times. |
| QUANTIZER_TORCH_PYTORCH_TRACE | Failed to get PyTorch traced graph from model and input arguments. The PyTorch internal failure reason is reported in message text. May needs adjust float model code. |
| QUANTIZER_TORCH_QUANT_CONFIG | Quantization configuration items are illegal. Refer to the message text. |
| QUANTIZER_TORCH_SHAPE_MISMATCH | Tensors shape are mismatched. Refer to the message text. |
| QUANTIZER_TORCH_VERSION | PyTorch version is not supported for the function or does not match vai_q_pytorch version (pytorch_nndct.__version__). Refer to the message text. |
| QUANTIZER_TORCH_XMODEL_BATCHSIZE | Batch size must be 1 when exporting XMODEL. |
| QUANTIZER_TORCH_INSPECTOR_OUTPUT_FORMAT | Inspector only supports dump SVG or PNG format. |
| QUANTIZER_TORCH_INSPECTOR_INPUT_FORMAT | Inspector no longer supports the fingerprint. Please provide architecture name instead. |
| QUANTIZER_TORCH_UNSUPPORTED_OPS | The quantization of the op is not supported. |
| QUANTIZER_TORCH_TRACED_NOT_SUPPORT | The model produced by 'torch.jit.script' is not supported in vai_q_pytorch. |
| QUANTIZER_TORCH_NO_SCRIPT_MODEL | vai_q_pytorch does not find any script model. |
| QUANTIZER_TORCH_REUSED_MODULE | The quantized module has been called multiple times in forward pass. If you want to share quantized parameters in multiple calls, call trainable_model with "allow_reused_module=True" |

表 54：**错误代码** (续)

| 错误代码 ID | 错误消息 |
|---|---|
| QUANTIZER_TORCH_DATA_PARALLEL_NOT_ALLOWED | torch.nn.DataParallel object is not allowed. |
| QUANTIZER_TORCH_INPUT_NOT_QUANTIZED | Input is not quantized. Please use QuantStub/DeQuantStub to define quantization scope. |
| QUANTIZER_TORCH_NOT_A_MODULE | Quantized operation must be instance of "torch.nn.Module". Replace the function by a "torch.nn.Module" object. Original source range is indicated in the message text. |
| QUANTIZER_TORCH_QAT_PROCESS_ERROR | Must call "trainable_model" first before getting deployable model. |
| QUANTIZER_TORCH_QAT_DEPLOYABLE_MODEL_ERROR | The given trained model has BN fused to CONV and cannot be converted to a deployable model. Make sure model.fuse_conv_bn() is not called. |
| QUANTIZER_TORCH_XMODEL_DEVICE | XMODEL can only be exported in CPU mode, use deployable_model(src_dir, used_for_xmodel=True) to get a CPU model. |
| WEGO_TORCH_UNKNOWN_ERROR | Unknown error |
| WEGO_TORCH_INTERNAL_ERROR | Internal error |
| WEGO_TORCH_INVALID_ARGUMENT | Invalid argument error |
| WEGO_TORCH_INVALID_MODEL | Invalid model error |
| WEGO_TORCH_OUT_OF_RANGE | Out of range error |
| WEGO_TORCH_UNIMPLEMENTED | Unimplemented error |
| WEGO_TORCH_RUNTIME_ERROR | Runtime error |
| XCOM_OP_CONV_PARAM_ERROR | Convolution parameter out of range or error, including feature map height, width, depth, channel, dilation size, transposition size, kernel height, kernel width, stride height, stride width or padding left, right, top, bottom, depth or fixed-point shift range or other network designed parameters. |
| XCOM_OP_IO_TENSOR_TYPE_ERROR | Error tensor type for IO operator such as load and save. |
| XCOM_OP_MEM_TYPE_ERROR | The op's output tensor's memory type is error. |
| XCOM_OP_PAD_SMF_MISSING | Failed to generate padding in pool since smf data missing. |
| XCOM_OP_POOL_SIZE_ERROR | Failed to calculate pooling size with formula. |
| XCOM_OP_SIGMOID_HEIGHT_NE | sigmoid operator need input and output have same height. |
| XCOM_OP_SIGMOID_WIDTH_NE | sigmoid operator need input and output have same width. |
| XCOM_OP_SIGMOID_CHANNEL_NE | sigmoid operator need input and output have same channel. |
| XCOM_OP_REORG_HEIGHT_NE | reorg operator need input height and output height have scale multiple. |
| XCOM_OP_REORG_WIDTH_NE | reorg operator need input width and output width have scale multiple. |
| XCOM_OP_REORG_CHANNEL_NE | reorg operator need input channel and output channel have scale multiple. |
| XCOM_OP_REORG_CHANNEL_OVERFLOW | feature channel size overflows regorg channel input. |
| XCOM_OP_TYPE_UNMATCH | unmatch operator type, it could be unknown operator or inappropriate suffix operator type. |
| XCOM_OP_TYPE_ERROR | op involved type error, unrecognized op involved type here. |
| XCOM_TILING_SIZE_ERROR | Tiling bank group size not enough and tiling failed. Perhaps input tensor or kernel size is too large or tiling bank aligned has calculation fault. |
| XCOM_DPUOP_DATA_SIZE_ERROR | Size not enough or unaligned while mapping smf onto banks with channel, width, depth, height, stride_h and other dimension incompatible. Please check bank info |

表 54：**错误代码** (续)

| 错误代码 ID | 错误消息 |
|---|---|
| XCOM_ACGEN_POOL_KERNEL_OUTRANGE | Pooling layer kernel size out of range. check network design or input data size. |
| XCOM_OP_NONLINEAR_TYPE_ERROR | Error of operator non-linear type. |
| XCOM_ACGEN_UNSUPPORT_QUANTIZATION | Unsupport quantization bit shift while assembly code generation. |
| XCOM_ACGEN_NONLINEAR_TYPE_ERROR | Unrecognized or unmatched type of non-linear type while assembly code generation |
| XCOM_ACGEN_BANK_IO_ERROR | Bank input or output addr count error while assembly code generation, it may exceed hardware capapcity. |
| XCOM_ACGEN_PRELU_ERROR | parameter-relu data info error while convolution assembly code generating, might be error with parameter data input width or number error. |
| XCOM_ACGEN_CONV_WEIGHTS_OC_NE | Convolution output channel number should be equal to convolution weights input, if not, check data life-circle. |
| XCOM_ACGEN_BANK_OC_WEIGHTS_UNALIGNED | Weights bank address need be aligned to convolution output channel. |
| XCOM_BANK_UNALIGNED_ADDRESSING | Trying to address in the middle of bank addr which is illegal, bank addressing only support aligned operation, for example, stride or address mod 16 == 0. |
| XCOM_ACGEN_CONV_FAKE_WEIGHTS_BANKID | Convolution weights input bank id need be equal to base bank id, check bank assignation of weights. |
| XCOM_ACGEN_CONV_FAKE_BIAS_BANKID | Convolution bias input bank id need be equal to base bank id, check bank assignation of bias. |
| XCOM_ACGEN_CONV_OCG_WEIGHTS_CNT_NE | Convolution weights input number need be equal to output channel group size, check weights input number. |
| XCOM_ACGEN_KERNEL_ALL_PAD | kernel are fulfilled with pad value, this is an unexpected situation, check kernel size and dilation value. |
| XCOM_ACGEN_BANK_ADDR_IN_OUTRANGE | Bank address input number need be ordered in hardware limitation. |
| XCOM_ACGEN_BANK_ADDR_OUT_OUTRANGE | Bank address output number need be ordered in hardware limitation. |
| XCOM_ACGEN_ELEW_IO_ERROR | Element wise operator have more than hardware capability input number, or, input and output number is not equal. check bank assignation. |
| XCOM_ACGEN_ELEW_IO_CHANNEL_NE | Element wise operator need input and output have same channel group size. |
| XCOM_ACGEN_ELEW_IO_LENGTH_NE | Element wise operator need input and output have same length. |
| XCOM_ACGEN_MUL_IO_ERROR | mul operator have more than hardware capability input number, or, input and output number is not equal. check bank assignation. |
| XCOM_ACGEN_INPUT_MISSING | Operator assembly code generation input bank address missing, check bank assignation. |
| XCOM_ACGEN_BLOB_MISSING | Blob shifting failed because cannot find specific blob id in blob area. check blob assignation. |
| XCOM_ACGEN_OUTPUT_MISSING | Operator assembly code generation ouptut bank address missing, check bank assignation. |
| XCOM_ACGEN_IO_TUPLE_NE | Some operator need input and output number be equal but here is not. check bank assignation. |
| XCOM_ACGEN_WEIGHTS_NOT_UNIQ | Some operator need uniq weights input bank, check bank assignation. |
| XCOM_ACGEN_PRELU_NOT_UNIQ | Some operator need uniq prelu input bank, check bank assignation. |
| XCOM_ACGEN_PARAM_NOT_UNIQ | Some operator need uniq param input bank like sigmoid, check bank aggregation. |

表 54：**错误代码** (续)

| 错误代码 ID | 错误消息 |
|---|---|
| XCOM_ACGEN_BIAS_NOT_UNIQ | Some operator need uniq bias input bank, check bank assignation. |
| XCOM_ACGEN_V3ME_BANK_MISSING | Operator's dest bank have no virtual bank or constant bank, on v3me conv operator at least need 1 type of bank. |
| XCOM_ACGEN_IC_WEIGHTS_CHANNEL_NE | Depth operator input channel and weights channel are not equal, check bank assignation or network structure. |
| XCOM_ACGEN_BIAS_WEIGHTS_CHANNEL_NE | Depth operator weighs channel and bias channel are not equal, check bank assignation. |
| XCOM_ACGEN_INVALIDE_STATUS | Compiler internal error, some status is invalid for assembly code generation like object was not inited, missing input or output data on dpu bank. check data flow and code logic. |
| XCOM_ACGEN_BANK_JUMP_READ_ERROR | The bank cannot jump read. |
| XCOM_ACGEN_BANK_JUMP_WRITE_ERROR | The bank cannot jump write. |
| XCOM_OP_ARGMAX_IO_HEIGHT_NE | argmax need input and output have equal height. |
| XCOM_OP_ARGMAX_IO_WIDTH_NE | argmax need input and output have equal width. |
| XCOM_OP_ARGMAX_IO_DEPTH_NE | argmax need input and output have equal depth. |
| XCOM_OP_ARGMAX_OC_NOT_UNIQ | argmax need output channel is 1. |
| XCOM_OP_CONCAT_IO_CHANNEL_NE | concat operator need input and output channel equal. |
| XCOM_OP_CORR_ELT_MUL_OUTPUT_ERROR | correlation eltwise multiply output depth calculate error. |
| XCOM_OP_CORR_ELT_MUL_IO_HEIGHT_NE | Correlation eltwise multiply need input height is equal to output height |
| XCOM_OP_CORR_ELT_MUL_IO_WIDTH_NE | Correlation eltwise multiply need input width is equal to output width |
| XCOM_OP_CORR_ELT_MUL_IO_CHANNEL_NE | Correlation eltwise multiply need input channel is equal to output channel |
| XCOM_OP_CORR_ELT_MUL_INPUT_CHANNEL_NE | Correlation eltwise multiply need all input channel are equal |
| XCOM_OP_COST_STRIDE_OUTPUT_DEPTH_NE | Cost operator need stride is equal to output depth |
| XCOM_OP_COST_IO_HEIGHT_NE | Cost operator need input and output have same height |
| XCOM_OP_COST_IO_WIDTH_NE | Cost operator need input and output have same width |
| XCOM_OP_COST_INPUT_DEPTH_NOT_UNIQ | Cost operator need input depth = 1 |
| XCOM_OP_COST_IO_CHANNEL_NE | Cost operator need input channel = 1/2 output channel |
| XCOM_OP_DOWNSAMPLE_IO_HEIGHT_NE | downsample need input height ceiling divide scale height equal to output height |
| XCOM_OP_DOWNSAMPLE_IO_WIDTH_NE | downsample need input width ceiling divide scale width equal to output width |
| XCOM_OP_DOWNSAMPLE_IO_CHANNEL_NE | downsample need input and output channel equal. |
| XCOM_OP_TDPTCONV3D_ICG_NOT_ENOUGH | Transposed depth conv 3d input channel group mult channel parallel is less than feature channel size. |
| XCOM_OP_TDPTCONV3D_KERNEL_HEIGHT_OVERFLOW | Transposed depth conv 3d kernel height overflow. |
| XCOM_OP_TDPTCONV3D_KERNEL_WIDTH_OVERFLOW | Transposed depthconv 3d kernel width overflow. |
| XCOM_OP_TDPTCONV3D_KERNEL_DEPTH_OVERFLOW | Transposed depth conv 3d kernel depth overflow. |
| XCOM_OP_TDPTCONV3D_STRIDE_HEIGHT_OVERFLOW | Transposed depth conv 3d stride height overflow. |
| XCOM_OP_TDPTCONV3D_STRIDE_WIDTH_OVERFLOW | Transposed depth conv 3d stride width overflow. |

表 54：错误代码 (续)

| 错误代码 ID | 错误消息 |
|---|---|
| XCOM_OP_TDPTCONV3D_STRIDE_DEPTH_OVERFLOW | Transposed depth conv 3d stride depth overflow. |
| XCOM_OP_TDPTCONV3D_OCG_NOT_ENOUGH | Transposed depth conv 3d output channel group mult channel parallel is less than feature channel size. |
| XCOM_OP_DPTCONV_IC_WEIGHT_DEPTH_NE | depth conv need kernel mult input channel group equal to weight bank depth |
| XCOM_OP_DPTCONV3D_KERNEL_HEIGHT_OVERFLOW | depth conv 3d kernel height overflow. |
| XCOM_OP_DPTCONV3D_KERNEL_WIDTH_OVERFLOW | depth conv 3d kernel width overflow. |
| XCOM_OP_DPTCONV3D_KERNEL_DEPTH_OVERFLOW | depth conv 3d kernel depth overflow. |
| XCOM_OP_DPTCONV3D_STRIDE_HEIGHT_OVERFLOW | depth conv 3d stride height overflow. |
| XCOM_OP_DPTCONV3D_STRIDE_WIDTH_OVERFLOW | depth conv 3d stride width overflow. |
| XCOM_OP_DPTCONV3D_STRIDE_DEPTH_OVERFLOW | depth conv 3d stride depth overflow. |
| XCOM_OP_DPTCONV3D_OCG_NOT_ENOUGHT | depth conv 3d output channel group mult channel parallel is less than feature channel size. |
| XCOM_OP_THRESHOLD_HEIGHT_NE | Threshold operator needs input and output have same height. |
| XCOM_OP_THRESHOLD_WIDTH_NE | Threshold operator needs input and output have same width. |
| XCOM_OP_THRESHOLD_CHANNEL_NE | Threshold operator needs input and output have same channel. |
| XCOM_OP_TILE_HEIGHT_NE | Tile operator needs input and output height have scale multiple relationship. |
| XCOM_OP_TILE_WIDTH_NE | Tile operator needs input and output width have scale multiple relationship. |
| XCOM_OP_TILE_CHANNEL_NE | Tile operator needs input and output channel have scale multiple relationship. |
| XCOM_OP_UPSAMPLE_HEIGHT_NE | upsample operator needs input and output height have scale multiple relationship. |
| XCOM_OP_UPSAMPLE_WIDTH_NE | upsample operator needs input and output width have scale multiple relationship. |
| XCOM_OP_UPSAMPLE_CHANNEL_NE | upsample operator needs input and output channel have scale multiple relationship. |
| XCOM_OP_ELEW_IO_CHANNEL_NE | element wise operator needs input and output channel equal |
| XCOM_OP_ELEW3D_IO_CHANNEL_NE | element wise 3d operator needs input and output channel equal |
| XCOM_OP_MUL_IO_HEIGHT_NE | mul operator needs input and output have same height. |
| XCOM_OP_MUL_IO_WIDTH_NE | mul operator needs input and output have same width. |
| XCOM_OP_MUL_IO_DEPTH_NE | mul operator needs input and output have same depth. |
| XCOM_OP_MUL_IO_CHANNEL_NE | mul operator needs input and output have same height. |
| XCOM_OP_MVR_IO_HEIGHT_NE | mvr operator needs input and output have same height. |
| XCOM_OP_MVR_IO_WIDTH_NE | mvr operator needs input and output have same width. |
| XCOM_OP_MVR_IO_DEPTH_NE | mvr operator needs input and output have same depth. |
| XCOM_OP_MVR_IO_CHANNEL_NE | mvr operator needs input and output have same height. |

表 54：**错误代码** (续)

| **错误代码 ID** | **错误消息** |
|---|---|
| XCOM_OP_CONV_KERNEL_WIDTH_OVERFLOW | Kernel width is larger than input width plus padding. that makes window cannot slide |
| XCOM_OP_CONV_KERNEL_HEIGHT_OVERFLOW | Kernel height is larger than input height plus padding. that makes window cannot slide, or, out of hardware limitation |
| XCOM_OP_CONV_STRIDE_WIDTH_OVERFLOW | Kernel width is larger than input width plus padding. that makes window cannot slide, or, out of hardware limitation |
| XCOM_OP_CONV_STRIDE_HEIGHT_OVERFLOW | Kernel height is larger than input height plus padding. that makes window cannot slide, or, out of hardware limitation |
| XCOM_OP_CONV_KERNEL_DEPTH_OVERFLOW | Kernel depth is larger than input height plus padding. that makes window cannot slide, or, out of hardware limitation |
| XCOM_OP_TCONV3D_KERNEL_HEIGHT_OVERFLOW | Kernel height is larger than input height plus padding. that makes window cannot slide, or, out of hardware limitation |
| XCOM_OP_TCONV3D_STRIDE_WIDTH_OVERFLOW | Kernel width is larger than input width plus padding. that makes window cannot slide, or, out of hardware limitation |
| XCOM_OP_TCONV3D_STRIDE_HEIGHT_OVERFLOW | Kernel height is larger than input height plus padding. that makes window cannot slide, or, out of hardware limitation |
| XCOM_OP_TCONV3D_KERNEL_DEPTH_OVERFLOW | Kernel depth is larger than input height plus padding. that makes window cannot slide, or, out of hardware limitation |
| XCOM_OP_TCONV3D_DILATION_HEIGHT_OVERFLOW | Dilation height is too large for input height |
| XCOM_OP_TCONV3D_DILATION_WIDTH_OVERFLOW | Dilation height is too large for input height |
| XCOM_OP_TCONV3D_DILATION_DEPTH_OVERFLOW | Dilation height is too large for input height |
| XCOM_OP_TCONV3D_ICG_WEIGHT_DEPTH_OVERFLOW | Input channel group stride overflows the weight bank depth. |
| XCOM_OP_CONV_DILATION_WIDTH_ALL_PAD | Padding width too large for dilation. Make all value in slide window are padding without input. |
| XCOM_OP_CONV_DILATION_HEIGHT_ALL_PAD | padding height too large for dilation, make all value in slide window are padding without input. |
| XCOM_OP_CONV_DILATION_DEPTH_ALL_PAD | padding depth too large for dilation, make all value in slide window are padding without input. |
| XCOM_OP_CONV_STRIDE_OVERFLOW | input channel stride overflow the weight bank depth. |
| XCOM_OP_TCONV3D_KENREL_DEPTH_OVERLARGE | kernel depth too large covering all feature input and padding, that makes window cannot slide. Or, out f hardware limitation. |
| XCOM_OP_TCONV3D_KENREL_WIDTH_OVERLARGE | kernel width too large covering all feature input and padding, that makes window cannot slide. Or, out f hardware limitation. |
| XCOM_OP_TCONV3D_KENREL_HEIGHT_OVERLARGE | kernel height too large covering all feature input and padding, that makes window cannot slide. Or, out f hardware limitation. |
| XCOM_OP_TCONV3D_DILATION_WIDTH_ALL_PAD | padding width too large for dilation, makes all value in slide window are padding without input feature. |
| XCOM_OP_TCONV3D_DILATION_HEIGHT_ALL_PAD | padding height too large for dilation, makes all value in slide window are padding without input feature. |
| XCOM_OP_TCONV3D_DILATION_DEPTH_ALL_PAD | padding depth too large for dilation, makes all value in slide window are padding without input feature. |
| XCOM_ACGEN_CONV_ERROR | error parameters while generating convolution, like some convolution parameter exceed hardware limitation or unexpected middle result generated. |

表 54：错误代码 (续)

| 错误代码 ID | 错误消息 |
|---|---|
| XCOM_BANK_CONV_ERROR | error banking status or behavior for convolution operation wile generating assembly code. check conv op data flow and tensor aggregation. |
| XCOM_BANK_INVALID_ID | invalid id while parsing for finding bank id. check bank name in target_factory. |
| XCOM_STR_PARSE_FAILED | Failed to parse specific string, perhaps it's an illegal string or empty string. |
| XCOM_DATA_SEGMENT_FAULT | data tensor or const tensor index exceed max tensor size, check index value. |
| XCOM_OP_CONFIG_MISSING | Failed to get specific op config. check op type config file. |
| XCOM_AIE_TARGET_INIT_FAILED | Failed to init aie target. |
| XCOM_AIE_SHIMTILE_OVERFLOW | aie tiling shim index or shim size out of range. |
| XCOM_AIE_MEMTILE_OVERFLOW | aie tiling memory index or memory size out of range. |
| XCOM_AIE_AIETILE_OVERFLOW | aie tiling index or bd index out of range. |
| XCOM_AIE_OUT_OF_BD | cannot find free bd for mem tiling. |
| XCOM_SLNODE_UNREGISTED | slnode target, type, name, or any info unregistered. |
| XCOM_INVOKE_BASE | An improper function invoking occurred! |
| XCOM_VALUE_UNMATCH | The value is not supposed! |
| XCOM_MEANINGLESS_VALUE | The value is meaningless. |
| XCOM_SIZE_UNMATCH | The object's size is not matching the requirement. |
| XCOM_OPERATOR_UNSUPPORT | This operator is not supposed! |
| XCOM_LEAF_SUBGRAPH_REQUIRED | Here requires a leaf subgraph. |
| XCOM_UNACCEPTABLE_SUBGRAPH | The subgraph is not allowed or meeting the requirements. |
| XCOM_PASS_MISS | Some compiler pass is missed. |
| XCOM_PASS_DEPENDENCY | Something wrong about pass dependency. |
| XCOM_DEBUGMANAGER_NOT_RECORDING | Invalid status for DebugManager recording. |
| XCOM_NO_PASS_RECORDED | No Pass has been recorded in DebugManager. |
| XCOM_DEBUGMANAGER_UNRECORDED_OP | Unrecorded op found. |
| XCOM_DDR_ADDR_ASSIGNMENT_FAILED | DDR address assignment is failed. |
| XCOM_DDR_PARAM_SPACE_INITIALIZTION_SIZE_ERROR | DDR parameter space initialization size error. Please contact us. |
| XCOM_UNIMPLEMENT | This part of function is unimplemented. |
| XCOM_UNDEFINED_STATE | This behavior is undefined. |
| XCOM_EXECUTE_SYSTEM_CMD_FAILED | Error occurred when execute the system command. |
| XCOM_TENSOR_DIMENSION_UNMATCH | The tensor dimension is unexpected. |
| XCOM_DATA_OUTRANGE | Data value is out of range! |
| XCOM_TYPE_UNMATCH | Unmatched type! |
| XCOM_ITEM_UNDEFINED | The requested item was not found or defined! |
| XCOM_OPERATION_FAILED | The supposed operation is failed! |
| XCOM_DIR_OPEN_FILE_FAILED | The file can't be read or can't access it. |
| XCOM_INVALID_GRAPH | Subgraph is null or error subgraph type like cpu subgraph using for dpu |

表 54：**错误代码** (续)

| 错误代码 ID | 错误消息 |
|---|---|
| XCOM_UNREGISTED_STRATEGY | This error code only used in dead code |
| XCOM_INVALID_ARCH_PARAM | This error code only used in dead code |
| XCOM_ACGEN_ERROR | Instruction generating fail, contact us. |
| XCOM_UNEXPECTED_VALUE | Inappropriate value at this place like nullptr or non-one value |
| XCOM_UNEXPECTED_ARCH | Unknown architecture name, check target_factory |
| XCOM_ARCH_UNREGISTED | Unknown architecture name, check target_factory |
| XCOM_ARCH_INVALID_NAME | Failed to file arch name in config dict, target factory or arch name serialization, check arch name. |
| XCOM_FILE_NOT_EXISTS | The file is not exists |
| XCOM_TARGET_REQUIRED | The compiler requires the target. |
| XCOM_GRAPH_REQUIRED | The compiler requires an input graph. |
| XCOM_LOGICAL_CONDITION_ERROR | The logical condition is wrong. |
| XCOM_INVALID_SUPERLAYER | The superlayer subgraph is invalid. |
| XCOM_UNSUPPORT_QUANTIZATION | The fix info is error or unsupported. |
| XCOM_SWIM_NODE_TYPE_ERROR | mismatch slnode type |
| XCOM_SWIM_OUTPUT_MISSING | swim lane output bank smf missing. |
| XCOM_SWIM_UNDEFINED_TYPE | undefined swim program or lane type. |
| XCOM_ALLOCATE_BANK_FAIL | XCompiler occurs error when allocating bank. Please contact us. |
| XCOM_TILING_FAIL | XCompiler occurs error when tiling. Please contact us. |
| XCOM_PM_FAIL | The compiler occurs an error when generating instructions, contact us. |
| XCOM_SUBGRAPH_ATTR_MISSING | The subgraph attribute is missing. |
| XCOM_ASSIGN_OUTPUT_OPS_FAILED | Assign output ops failed. |
| XCOM_DDR_REG_ID_SIZE_UNMATCH | The DDR reg id size unmatch. Please contact us. |
| XCOM_DDR_OPTIMIZATION_0_FAILED | DDR assignment optimization failed (code 0). Please contact us. |
| XCOM_DDR_OPTIMIZATION_1_FAILED | DDR assignment optimization failed (code 1). Please contact us. |
| XCOM_TRANSPOSED_CONV_WEIGHTS_DDR_OPTIMIZATION_ERROR | DDR assignment optimization failed during optimizing the transposed convolution's weights. Please contact us. |
| XCOM_DIRECTORY_EXIST | The directory is existing, can't be created multiple times. |
| XCOM_DIRECTORY_NOT_EXIST | The directory is not existing. |
| XCOM_INVALID_COMPILE_MODE | The compile mode is not supported now. |
| XCOM_INVALID_TARGET | Invalid DPU target |
| XCOM_DPU_MEMORY_ALLOCATION_FAILED | error mapping smfs onto dpu banks since error input group of smfs like no specific type (data, const data (weights, bias ...)) found in given Smf group. Or unaligned smf info on data width, height or their stride, channel and channel group stride. Unaligned smfs cannot be aggregated on aggregation dimension. |
| XCOM_UNSUPPORT_NONLIEAR_TYPE | The nonlinear type is unsupported by DPU. |
| XCOM_PAD_KERNEL_SIZE_UNMATCH | The pad size is not correct comparing with the kernel size. |
| XCOM_DATA_DEPENDCY_MISSING | Generate code failed. |
| XCOM_MULTIPLE_WEIGHTS | There are more than one weights for some ops. |
| XCOM_MULTIPLE_BIAS | There are more than one bias for some ops. |

表 54：**错误代码** (续)

| 错误代码 ID | 错误消息 |
|---|---|
| XCOM_MULTIPLE_PRELU | There are more than oen prelu for some ops. |
| XCOM_TOO_MANY_INPUTS | There are too many inputs for the op. |
| XCOM_TENSOR_SHAPE_UNMATCH | Tensor shapes for some ops are not matching. |
| XCOM_UNSUPPORT_KERNEL_SIZE | The op's kernel size is not supported. |
| XCOM_CODE_GEN_ERROR | Code generation fail. |
| XCOM_UNSUPPORT_ROUND_MODE | The round mode is not supported. |
| XCOM_ADDITION_OVERFLOW | The addition is overflow. |
| XCOM_INT_COMPOSITION_INVALID_RANGE | The integer composition's output range constraint is invalid. Please contact us. |
| XCOM_TO_XINT_DATA_SIZE_UNMATCH | The input data vector's size is unmatching with the xint's bit width |
| XCOM_REVERT_XINT_DATA_SIZE_UNMATCH | The input data vector's size is unmatching with the xint's original bit width. |
| XCOM_DWCONV_PARAM_REORDER_SIZE_UNMATCH | size unmatching occurred during reordering the DepthwiseConv2d's parameter. |
| XCOM_CONV_PARAM_REORDER_SIZE_UNMATCH | size unmatching occured during reordering the Conv2d's parameter. |
| XCOM_AIE_CORE_NUM_MISMATCH | The config of aie core number mismatchs. |
| XCOM_AIE_TIMING_CONFIG_MISMATCH | The config inside aie timing calculating mismatchs. |
| XCOM_AIE_TILING_FAIL | There is not enough bank space inside AIE local memory for this tensor |
| XCOM_AIE_UNSUPPORTED_OP | Unsupported op for aie tiling |
| XCOM_PARTITIONENGINE_HINTS_ERROR | The partition engine's hints are invalid. Please contact us. |
| XCOM_PARSE_FAIL | Failed to parse structured data! |
| XCOM_PARTITION_REPEATED_REGISTRATION | Repeated checker registration for the op_type and arch_type in partition. |
| XCOM_UNREGISTED_SLNODE | Unregisted Slnode |
| XCOM_GET_CHANNEL_FAILED | xGet channel failed. |
| XCOM_GET_PACKET_ID_FAILED | xGet packet id failed. |
| XCOM_GET_PACKET_TYPE_FAILED | xGet packet type failed. |
| XCOM_GET_LOCK_ID_FAILED | xGet lock id failed. |
| XCOM_GET_BD_FAILED | xGet bd failed. |
| XCOM_SMF_SPEC_MISSING | no found such spec for the smf |
| XCOM_SMF_MISSING | no found such smf |
| XCOM_SMF_Y_SIZE_ERROR | smf y direction size overflow bank height with padding. |
| XCOM_SMF_C_SIZE_ERROR | channel direction smf need height = 1, width = 1, pad top and bottom = 0. |
| XCOM_SMF_COORDINATE_ERROR | smf on coordinate have error size or missing. |
| XCOM_SMF_CONCAT_ERROR | smf on concatenate have error size or missing. |
| XCOM_SMF_BIAS_ERROR | bias smf size error or missing. |
| XCOM_SMF_PARAM_ERROR | param smf size error or missing. |
| XCOM_SMF_TYPE_ERROR | error smf type. |
| XCOM_SMF_TENSOR_TYPE_ERROR | tensor type error while smf arrangment |
| XCOM_SMF_RESERVED_ERROR | smf dynamic size error or missing or addressing failed. |
| XCOM_SMF_BANK_MANAGEMENT_ERROR | error happens in bank management, error name addressing or missing. |

表 54：**错误代码** (续)

| 错误代码 ID | 错误消息 |
|---|---|
| XCOM_BANK_SMF_EXIST | smf name already exists in bank. |
| XCOM_BANK_ADDR_MISSING | bank addressing missing. |
| XCOM_BANK_ADDR_OVERFLOW | bank addressing overflow. |
| XCOM_BANK_ADDR_ERROR | bank block addr have error sequence or non-exist addressing. |
| XCOM_USER_FILE_NOT_EXISTS | The file does not exist |
| XCOM_USER_FILE_OPERATION_ERR | The file operation failed |
| XCOM_USER_INVALID_COMPILE_MODE | The compile mode is not supported now. |
| XCOM_USER_DIRECTORY_NOT_EXIST | The directory does not exist, create it first. |
| XCOM_USER_DIRECTORY_ALREADY_EXIST | The directory already exist, remove it first. |
| XCOM_USER_INVALID_CMD_PARAM | Invalid cmdline parameter |
| XCOM_USER_INVALID_TARGET | Invalid DPU target is given by cmdline. |
| XCOM_USER_INVALID_OUTPUT_OPS | The output ops user specified can't be found in the network. Please check the op names. |
| XCOM_USER_INVALID_OUTPUT_TENSORS | The output tensors user specified can't be found in the network. Please check the tensor names. |
| XCOM_USER_UNSUPPORTED_SYSTEM | The function is not supported by current operating system. |
| XCOM_PASS_DEPENDENCY_ERROR | Something wrong about pass dependency. |
| XCOM_PASS_UNREGISTERED | Pass has not been registered. |
| XCOM_PASS_NULL_POINTER | Accessing of uninitialized object or pointer |
| XCOM_PASS_TARGET_UNIMPLEMENTED | The target has not been implemented yet. |
| XCOM_PASS_GRAPH_ATTR_MISSING | Necessary attribution has not been set for the graph. |
| XCOM_PASS_OP_INVALID_ATTR | The parameter of the operator is invalid. Please check the input network. |
| XCOM_PASS_OP_ATTR_MISMATCH | The parameter of the operator is unexpected. |
| XCOM_PASS_INVALID_BLOB_NUMBER | Blob number of the operator is invalid. Please check the input network. |
| XCOM_PASS_OP_ATTR_MISSING | The requisite parameter is missing. Please check input network. |
| XCOM_PASS_TENSOR_SIZE_ERROR | Size of the tensor is invalid. Please check input network. |
| XCOM_PASS_TENSOR_SIZE_MISMATCH | Size mismatch between correlative tensors. |
| XCOM_PASS_TENSOR_DIMS_MISMATCH | Dimensions of the tensor is unexpected. |
| XCOM_COMGRAPH_OP_MISSING | The op is missing in specific graph. |
| XCOM_COMGRAPH_OP_CONNECTION_MISSING | The connection is missing in the graph. |
| XCOM_COMGRAPH_OP_CONNECTION_INVALID | The connection between two specific op is unexpected. Check input network. |
| XCOM_COMGRAPH_ATTR_NOT_ASSIGNED | The requisite attribution of xcomgraph has not be assigned. |
| XCOM_COMGRAPH_GRAPH_INVALID_STRUCTURE | There is a unexpected pattern in the graph. Please check the input network. |
| XCOM_COMGRAPH_SUBGRAPH_MISSING | The subgraph is missing. |
| XCOM_COMGRAPH_BANK_UNEXPECTED_STATE | Bank assignment is rejected by a particular subgraph. |
| XCOM_COMGRAPH_BANK_INFO_MISMATCH | Bank requirement mismatch between 2 ops. |
| XCOM_FRONTEND_SUBGRAPH_MISSING | The subgraph is missing. |
| XCOM_FRONTEND_NULL_POINTER | Accessing of uninitialized object or pointer |

表 54：**错误代码** (续)

| 错误代码 ID | 错误消息 |
|---|---|
| XCOM_FRONTEND_UNEXPECTED_STATE | A impossible state occurs. There might be a logical error of programming. |
| XCOM_FRONTEND_GRAPH_OP_MISSING | The op is missing in specific graph. |
| XCOM_FRONTEND_GRAPH_TEMPLATE_MISMATCH | Pattern mismatch between template and replacing process. |
| XCOM_FRONTEND_GRAPH_LEVEL_MISMATCH | A unsuitable level of graph is given for current function. |
| XCOM_FRONTEND_GRAPH_ATTR_MISSING | Necessary attribution has not been set for the graph. |
| XCOM_FRONTEND_GRAPH_INVALID_STRUCTURE | There is a unexpected pattern in the graph. Please check the input network. |
| XCOM_FRONTEND_OP_INVALID_ATTR | The parameter of the operator is invalid. Please check the input network. |
| XCOM_FRONTEND_OP_INVALID_BLOB_NUMBER | Blob number of the operator is invalid. Please check the input network. |
| XCOM_FRONTEND_OP_UNSUPPORTED_BLOB_NUMBER | Blob number of the operator is unsupported by target DPU. |
| XCOM_FRONTEND_OP_UNSUPPORTED_ATTR | An attribution of the operator is unsupported by target DPU. |
| XCOM_FRONTEND_OP_UNSUPPORTED_MODE | The mode has not been implemented for specific operator. |
| XCOM_FRONTEND_OP_UNSUPPORTED_NONLIEAR | The nonlinear(or activation) is unsupported by target DPU |
| XCOM_FRONTEND_OP_UNSUPPORTED | The operator has not been implemented by target DPU. |
| XCOM_FRONTEND_OP_ATTR_MISMATCH | The parameter of the operator is unexpected. |
| XCOM_FRONTEND_QUANT_ATTR_MISSING | Quantizing information for the op is mission |
| XCOM_FRONTEND_QUANT_ATTR_OUT_OF_RANGE | The shiftbit for quantizing is out of range, that the range is restricted by target DPU. |
| XCOM_FRONTEND_TENSOR_DIMS_MISMATCH | Dimensions of the tensor is unexpected. |
| XCOM_FRONTEND_TENSOR_SHAPE_MISMATCH | Shape mismatch between 2 correlative tensors. |
| XCOM_FRONTEND_TENSOR_SIZE_MISMATCH | Size mismatch between 2 correlative tensors. |
| XCOM_FRONTEND_DATA_TYPE_MISMATCH | Data type mismatch between 2 correlative tensors. |
| XCOM_FRONTEND_BITWIDTH_MISMATCH | The bit width of the tensor is unexpected. |
| XCOM_GENINST_NULL_POINTER | Accessing of uninitialized object or pointer |
| XCOM_GENINST_INVALID_VALUE | A invalid value is given for specific function. |
| XCOM_GENINST_GRAPH_INVALID_STRUCTURE | There is a unexpected pattern in the graph. Please check the input network. |
| XCOM_GENINST_OP_UNSUPPORTED | The operator has not been implemented by target dpu. |
| XCOM_GENINST_OP_UNSUPPORTED_MODE | The mode has not been implemented for specific operator. |
| XCOM_GENINST_OP_UNSUPPORTED_NONLIEAR | The nonlinear(or activation) is unsupported by target DPU |
| XCOM_GENINST_OP_INVALID_BLOB_NUMBER | Blob number of the operator is invalid. Please check the input network. |
| XCOM_GENINST_TARGET_UNIMPLEMENTED | The target has not been implemented yet. |
| XCOM_GENINST_TARGET_BANK_ERR | No output bank associated with current op within target DPU. |
| XCOM_GENINST_GRAPH_TEMPLATE_MISMATCH | Pattern mismatch between template and replacing process. |
| XCOM_GENINST_BANK_INFO_MISMATCH | Bank requirement mismatch between 2 ops. |
| XCOM_GENINST_TILING_FAIL | Failed to get a available tiling scheme. |
| XCOM_GENINST_CODE_GEN_ERROR | Code generation fail. |
| XCOM_OPFACTORY_ILLEGAL_NAME | The object name is illegal. |

表 54：**错误代码** (续)

| 错误代码 ID | 错误消息 |
|---|---|
| XCOM_OPFACTORY_UNEXPECTED_STATE | A impossible state occurs. There might be a logical error of programming. |
| XCOM_OPFACTORY_GRAPH_INVALID_STRUCTURE | There is a unexpected pattern in the graph. Please check the input network. |
| XCOM_OPFACTORY_OP_INVALID_BLOB_NUMBER | Blob number of the operator is invalid. Please check the input network. |
| XCOM_OPFACTORY_OP_UNSUPPORTED_BLOB_NUMBER | Blob number of the operator is unsupported by target DPU. |
| XCOM_OPFACTORY_OP_UNSUPPORTED_MODE | The mode has not been implemented for specific operator. |
| XCOM_OPFACTORY_OP_UNSUPPORTED_NONLIEAR_TYPE | The nonlinear(or activation) is unsupported by target DPU |
| XCOM_OPFACTORY_OP_UNSUPPORTED | The operator has not been implemented by target DPU. |
| XCOM_OPTIMIZE_GRAPH_INVALID_STRUCTURE | There is a unexpected pattern in the graph. Please check the input network. |
| XCOM_OPTIMIZE_GRAPH_OPERATION_FAILED | A graphic problem has caused by graphic operation. |
| XCOM_OPTIMIZE_OP_UNSUPPORTED_ATTR | An attribution of the operator is unsupported by target DPU. |
| XCOM_OPTIMIZE_OP_INVALID_BLOB_NUMBER | Blob number of the operator is invalid. Please check the input network. |
| XCOM_OPTIMIZE_TENSOR_SHAPE_INVALID | Tensor shape is invalid. Please check the input network. |
| XCOM_OPTIMIZE_TARGET_BANK_ERR | No output bank associated with current op within target DPU. |
| XCOM_OPTIMIZE_OP_ATTR_MISMATCH | The parameter of the operator is unexpected. |
| XCOM_UTIL_NULL_POINTER | Accessing of uninitialized object or pointer |
| XCOM_UTIL_OPERATION_DENIED | The operation is not permitted. |
| XCOM_UTIL_PARSE_FAIL | Failed to parse structured data. |
| XCOM_UTIL_OP_UNSUPPORTED_NONLIEAR_TYPE | The nonlinear(or activation) is unsupported by target DPU |
| XCOM_UTIL_OP_UNSUPPORTED | The operator has not been implemented by target DPU. |
| XCOM_UTIL_TENSOR_DIMS_MISMATCH | Dimensions of the tensor is unexpected. |
| XCOM_UTIL_TENSOR_SHAPE_INVALID | Tensor shape is invalid. Please check the input network. |
| XCOM_UTIL_DATA_TYPE_INVALID | Data type is invalid for current context. |
| XCOM_UTIL_ATTR_OUT_OF_RANGE | The value is out of range. |
| XCOM_UTIL_INVALID_VALUE | A invalid value is given for specific function. |
| XCOM_UTIL_INVALID_VALUE_RANGE | The data range is illegal. |
| XCOM_UTIL_ADDITION_OVERFLOW | The addition is overflow. |
| XCOM_UTIL_DATA_SIZE_MISMATCH | Size mismatch between 2 data chunk. |
| XCOM_UTIL_BANK_ALLOC_FAILED | Failed to get a available scheme of bank assignment. |
| XCOM_BANKASSIGN_GRAPH_INVALID_STRUCTURE | There is a unexpected pattern in the graph. Please check the input network. |
| XCOM_BANKASSIGN_INVALID_ITEM | The item is not available yet. |
| XCOM_BANKASSIGN_OUT_OF_BANK_SIZE | Insufficiency of bank size, input model might be too large. |
| XCOM_BANKASSIGN_TARGET_ENGINE_UNREGISTERED | The engine is not registered for dpu target. |
| XCOM_BANKASSIGN_INVALID_VALUE | A invalid value is given for specific function. |
| XCOM_BANKASSIGN_TARGET_BANK_ERR | No output bank associated with current op within target DPU. |

表 54：**错误代码** (续)

| 错误代码 ID | 错误消息 |
|---|---|
| XCOM_BANKASSIGN_BANK_UNEXPECTED_STATE | Bank space manager might be in disorder. |
| XCOM_BANKASSIGN_OP_INVALID_BLOB_NUMBER | Blob number of the operator is invalid. Please check the input network. |
| XCOM_BANKASSIGN_BANK_INFO_MISMATCH | Bank requirement mismatch between 2 ops. |
| XCOM_PARTITION_GRAPH_INVALID_STRUCTURE | There is a unexpected pattern in the graph. Please check the input network. |
| XCOM_PARTITION_NULL_POINTER | Accessing of uninitialized object or pointer |
| XCOM_DDRALLOC_NULL_POINTER | Accessing of uninitialized object or pointer. |
| XCOM_DDRALLOC_UNEXPECTED_STATE | A impossible state occurs. There might be a logical error of programming. |
| XCOM_DDRALLOC_INVALID_ITEM | Failed to get a available scheme of bank assignment. |
| XCOM_DDRALLOC_ITEM_UNDEFINED | The item has not be defined. |
| XCOM_DDRALLOC_DATA_SIZE_MISMATCH | Size mismatch between 2 data chunk. |
| XCOM_DDRALLOC_MEM_ACCESS_OVERFLOW | Memory access overflow. |
| XCOM_DDRALLOC_OUT_OF_MEM | DDR space is not enough for current model. |
| XCOM_DDRALLOC_FEATURE_UNIMPLEMENT | The feature of DDR allocating optimization is not implemented. |
| XCOM_DDRALLOC_OPERATION_DENIED | The operation is not permitted. |
| XCOM_DDRALLOC_TARGET_UNIMPLEMENTED | The target has not been implemented yet. |
| XCOM_DDRALLOC_ASSIGNMENT_STATE_ERROR | The state of DDR allocation is unexpected. |
| XCOM_DDRALLOC_PARAM_SPACE_INITIALIZTION_SIZE_ERROR | DDR parameter space initialization size error. Please contact us. |
| XCOM_DDRALLOC_GRAPH_LEVEL_MISMATCH | A unsuitable level of graph is given for current function. |
| XCOM_DDRALLOC_GRAPH_OP_MISSING | The op is missing in specific graph. |
| XCOM_DDRALLOC_GRAPH_INVALID_STRUCTURE | There is a unexpected pattern in the graph. Please check the input network. |
| XCOM_DDRALLOC_OP_UNSUPPORTED | The operator has not been implemented by target DPU. |
| XCOM_DDRALLOC_OP_UNSUPPORTED_MODE | The mode has not been implemented for specific operator. |
| XCOM_DDRALLOC_OP_INVALID_BLOB_NUMBER | Blob number of the operator is invalid. Please check the input network. |
| XCOM_DDRALLOC_OP_UNSUPPORTED_NONLIEAR_TYPE | The nonlinear(or activation) is unsupported by target DPU |
| XCOM_DDRALLOC_TENSOR_DIMS_MISMATCH | Dimensions of the tensor is unexpected. |
| XCOM_DDRALLOC_TENSOR_SHAPE_MISMATCH | Shape mismatch between 2 correlative tensors. |
| XCOM_DDRALLOC_TENSOR_SIZE_ERROR | Size of the tensor is invalid. Please check input network. |
| XCOM_DDRALLOC_TENSOR_SIZE_MISMATCH | Size mismatch between 2 correlative tensors. |
| VAILIB_DPU_TASK_NOT_FIND | Model files not find |
| VAILIB_DPU_TASK_OPEN_ERROR | Open file failed |
| VAILIB_DPU_TASK_CONFIG_PARSE_ERROR | Parse model config file failed |
| VAILIB_DPU_TASK_TENSORS_EMPTY | Runner has no input tensors |
| VAILIB_DPU_TASK_SUBGRAPHS_EMPTY | Runner has no subgraphs |
| VAILIB_CPU_RUNNER_OPEN_LIB_ERROR | dlopen can not open lib |
| VAILIB_CPU_RUNNER_LOAD_LIB_SYM_ERROR | dlsym load symbol error |

表 54：**错误代码** (续)

| 错误代码 ID | 错误消息 |
|---|---|
| VAILIB_CPU_RUNNER_TENSOR_BUFFER_NOT_FI ND | Can not find tensor buffer with this name |
| VAILIB_CPU_RUNNER_TENSOR_BUFFER_NOT_CO NTINOUS | Tensor buffer not continuous |
| VAILIB_CPU_RUNNER_READ_FILE_ERROR | Fail to read file |
| VAILIB_CPU_RUNNER_WRITE_FILE_ERROR | Fail to write file |
| VAILIB_CPU_RUNNER_CPU_OP_NOT_FIND | Can not find op with this name |
| VAILIB_GRAPH_RUNNER_NOT_FIND | GraphTask can not find tensor or tensor buffer |
| VAILIB_GRAPH_RUNNER_DPU_BATCH_ERROR | GraphTask get dpu batch not equal |
| VAILIB_GRAPH_RUNNER_NOT_SUPPORT | The function or value are not supported in graph runner |
| VAILIB_GRAPH_RUNNER_NOT_OVERRIDE | The function has not been overridden |
| VAILIB_MATH_NOT_SUPPORT | The function or value are not supported in vai-math |
| VAILIB_MATH_FIX_POS_ERROR | Softmax table not support the fix position value |
| VAILIB_MODEL_CONFIG_NOT_FIND | Model config info not find |
| VAILIB_MODEL_CONFIG_OPEN_ERROR | Model config file or directory open error |
| VAILIB_MODEL_CONFIG_CONFIG_PARSE_ERROR | Model config file parse error |
| VAILIB_BENCHMARK_LIST_EMPTY | Can not found images. List of images are empty |
| VAILIB_DEMO_CANVAS_ERROR | Canvas image size is too small |
| VAILIB_DEMO_GST_ERROR | Failed to open gstreamer |
| VAILIB_DEMO_IMAGE_LOAD_ERROR | Failed to load image |
| VAILIB_DEMO_OUT_OF_BOUNDARY | Gui rects out of boundary |
| VAILIB_DEMO_VIDEO_OPEN_ERROR | Cannot open video stream |
| VART_OPEN_DEVICE_FAIL | Cannot open device |
| VART_LOAD_XCLBIN_FAIL | Bitstream download failed |
| VART_LOCK_DEVICE_FAIL | Cannot lock device |
| VART_FUNC_NOT_SUPPORT | Function not support |
| VART_XMODEL_ERROR | XMODEL error |
| VART_GRAPH_ERROR | Graph error |
| VART_TENSOR_INFO_ERROR | Tensor info error |
| VART_DPU_INFO_ERROR | DPU info error |
| VART_SYSTEM_ERROR | File system error |
| VART_DEVICE_BUSY | Device busy |
| VART_DEVICE_MISMATCH | Device mismatch |
| VART_DPU_ALLOC_ERROR | DPU allocate error |
| VART_VERSION_MISMATCH | Version mismatch |
| VART_OUT_OF_RANGE | Array index out of range |
| VART_SIZE_MISMATCH | Array size not match |
| VART_NULL_PTR | Nullptr |
| VART_XRT_NULL_PTR | Nullptr |
| VART_XRT_DEVICE_BUSY | Device busy |

表 54：**错误代码** (续)

| 错误代码 ID | 错误消息 |
|---|---|
| VART_XRT_READ_ERROR | Read error |
| VART_XRT_READ_CU_ERROR | Read cu fatal |
| VART_XRT_FUNC_FAULT | XRT function fault |
| VART_XRT_DEVICE_AVAILABLE_ERROR | No devices available |
| VART_XRT_CU_AVAILABLE_ERROR | No CU available |
| VART_XRT_OPEN_CONTEXT_ERROR | xclOpenContext failed |
| VART_XRM_CREATE_CONTEXT_ERROR | failed to create XRM context |
| VART_XRM_CONNECT_ERROR | Failed to connect to XRM |
| VART_XRM_ACQUIRE_CU_ERROR | Could not acquire CU |
| VART_DEVICE_BUFFER_ALLOC_ERROR | Cannot alloc device buffer -- unknown datatype |
| VART_XCLBIN_READ_ERROR | Failed to open xclbin file for reading |
| VART_XCLBIN_DOWNLOAD_ERROR | Bitstream download failed ! |
| VART_CONTROLLER_VIR_MEMORY_ALLOC_ERROR | not enough virtual space on host |
| VART_VERSION_MISMATCH_ERROR | subgraph's version is mismatch with xclbin |
| VART_CONTROLLER_DUMP_FOLDER_CREATE_ERROR | Create dump folder error |
| VART_CONTROLLER_DUMP_SUBFOLDER_CREATE_ERROR | Create sub dump folder error |
| VART_DEVICE_MEMORY_ALLOC_ERROR | Device memory not enough, alloc fail |
| VART_TENSOR_BUFFER_CREATE_ERROR | TensorBuffer create fail |
| VART_TENSOR_BUFFER_INVALID | invalid tensorbuffer input or output |
| VART_DPU_EXEC_ERROR | DPU run fail |
| VART_DPU_TIMEOUT_ERROR | DPU timeout |
| VART_CONTROLLER_DUMP_ERROR | dump failed |
| VART_XCLBIN_PATH_INVALID | xclbinPath is not set, consider setting XLNX_VART_FIRMWARE. |
| VART_GRAPH_FINGERPRINT_ERROR | no hardware info in subgraph |
| VART_TENSOR_BUFFER_CHECK_ERROR | TensorBuffer size less than offset, input shape invalid |
| VART_TENSOR_BUFFER_DIMS_ERROR | input dims shape is invalid |
| XIR_ACCESS_ADDRESS_OVERFLOW | The address you try to access does not exsit! |
| XIR_ADD_OP_FAIL | Failed to add an op! |
| XIR_FILE_NOT_EXIST | File doesn’t exist! |
| XIR_INTERNAL_ERROR | it is an internal bug supposed never happen |
| XIR_INVALID_ARG_OCCUR | Invalid arg occurrence! |
| XIR_INVALID_ATTR_DEF | Invalid attribute definition! |
| XIR_INVALID_ATTR_OCCUR | Invalid attr occurrence! |
| XIR_INVALID_DATA_TYPE | The data type is invalid. |
| XIR_MEANINGLESS_VALUE | The value you set for this parameter makes no sense. |
| XIR_MULTI_DEFINED_OP | Multiple definition of OP! |
| XIR_MULTI_DEFINED_TENSOR | Multiple definition of Tensor! |

表 54：**错误代码** (续)

| 错误代码 ID | 错误消息 |
|---|---|
| XIR_MULTI_REGISTERED_ARG | Multiple registration of argument! |
| XIR_MULTI_REGISTERED_ATTR | Multiple registration of attribute! |
| XIR_MULTI_REGISTERED_EXPANDED_ATTR | Multiple registration of static attr! |
| XIR_MULTI_REGISTERED_OP | Multiple registration of operator! |
| XIR_OPERATION_FAILED | Fail to execute command! |
| XIR_OP_DEF_SHAPE_HINT_MISSING | A shape hint is required by the op definition |
| XIR_OP_NAME_CONFLICT | There are at least two ops assigned the same name, check the ops' name. |
| XIR_OUT_OF_RANGE | idx out of range! |
| XIR_PROTECTED_MEMORY | The content in protected momory for tensor can not be modified! |
| XIR_READ_PB_FAILURE | failed to read pb file |
| XIR_REMOVE_OP_FAIL | Failed to remove an op! |
| XIR_SHAPE_UNMATCH | The shape is unmatching. |
| XIR_SUBGRAPH_ALREADY_CREATED_ROOT | Already created root subgraph for the graph! |
| XIR_SUBGRAPH_CREATE_CHILDREN_FOR_NONLEAF | |
| XIR_SUBGRAPH_HAS_CYCLE | Children from a same subgraph depend each other! |
| XIR_SUBGRAPH_INVALID_MERGE_REQUEST_NONCHILD | Cannot merge subgraphs which are not children! |
| XIR_SUBGRAPH_INVALID_MERGE_REQUEST_NONLEAF | Cannot merge subgraphs which are not leaves! |
| XIR_UNDEFINED_ATTR | Undefined attribute! |
| XIR_UNDEFINED_INPUT_ARG | Undefined input arg! |
| XIR_UNDEFINED_OP | Access undefined OP! |
| XIR_UNEQUIVALENT_ATTRIBUTE | These two attributes/parameters are not equivalent. |
| XIR_UNEXPECTED_VALUE | Unexpected value! |
| XIR_UNKNOWNTYPE_TENSOR | The datatype of this tensor is not specified. |
| XIR_UNREGISTERED_ARG | Unregistered argument! |
| XIR_UNREGISTERED_ATTR | Unregistered attribute! |
| XIR_UNREGISTERED_OP | Unregistered operator! |
| XIR_UNSUPPORTED_ROUND_MODE | unsupported round mode. |
| XIR_UNSUPPORTED_TYPE | unsupported data type for attr value |
| XIR_VALUE_UNMATCH | Value unmatch! |
| XIR_WRITE_PB_FAILURE | failed to write pb file |
| XIR_XIR_UNDEFINED_OPERATION | Undefined operation! |
| TARGET_EXPLORER_XCLBIN_ERROR | No xclbin specified |
| TARGET_EXPLORER_XCLBIN_ENV_ERROR | DPU xclbin path specified by ‘XLNX_VART_FIRMWARE’ not exist, check! |
| TARGET_EXPLORER_XCLBIN_ENV_VAL_ERROR | ‘XLNX_VART_FIRMWARE’ need to be specified like /path/to/xxx.xclbin, check! |
| TARGET_EXPLORER_SYS_DEVICE_CHECK_ERROR | The system has no device |
| TARGET_EXPLORER_XCLBIN_SET_ERROR | xclbinPath is not set, consider setting XLNX_VART_FIRMWARE. |

表 54：**错误代码** (续)

| 错误代码 ID | 错误消息 |
|---|---|
| TARGET_EXPLORER_NO_DPU_ERROR | xclbin is not for DPU, can't find DPU kernel in xclbin |
| TARGET_EXPLORER_BATCH_ERROR | Only support multiple DPU cores with same batch and fingerprint |
| TARGET_EXPLORER_DEVICE_CHECK_ERROR | No device available for current xclbin |
| TARGET_FACTORY_INVALID_ARCH | Invalid target arch! |
| TARGET_FACTORY_INVALID_FEATURE_CODE | Invalid target feature code! |
| TARGET_FACTORY_INVALID_ISA_VERSION | Invalid target ISA version! |
| TARGET_FACTORY_INVALID_TYPE | Invalid target type! |
| TARGET_FACTORY_MULTI_REGISTERED_TARGET | Multiple registration of target! |
| TARGET_FACTORY_PARSE_TARGET_FAIL | Fail to parse target from prototxt! |
| TARGET_FACTORY_UNREGISTERED_TARGET | Unregistered target! |

附录 B

# 附加资源与法律声明

## 查找其他文档

**文档门户**

AMD 自适应计算文档门户是旨在使用您的网页浏览器提供健全的文档搜索和导航的在线工具。要访问文档门户，请转至 https://docs.xilinx.com。

**注释：**单击链接将打开英语版本，但您可从下拉列表中选择简体中文版本（如可用）。请注意，简体中文版本可能比英语版本旧。

**Documentation Navigator**

Documentation Navigator (DocNav) 是预安装的工具，支持访问 AMD 自适应计算文档、视频和支持资源，您可在其中通过筛选和搜索来查找信息。要打开 DocNav，请执行以下操作：

- 在 AMD Vivado™ IDE 中，单击“Help” → “Documentation and Tutorials”。
- 在 Windows 上，单击“Start”（开始）按钮并选中“Xilinx Design Tools” → “DocNav”。
- 在 Linux 命令提示中输入 `docnav`。

**注释：**如需了解有关 DocNav 的更多信息，请参阅《Documentation Navigator 用户指南》(UG968)。

**注释：**您无法从 DocNav 访问简体中文版本。请使用设计中心网页。

**设计中心 (Design Hub)**

AMD 设计中心提供了根据设计任务和其他主题整理的文档链接，可供您用于了解关键概念以及常见问题解答。要访问设计中心，请执行以下操作：

- 在 DocNav 中，单击“Design Hubs View”选项卡。
- 转至设计中心网页。

## 支持资源

如需获取答复记录、技术文档、下载以及论坛等支持资源，请访问技术支持。

# 参考资料

以下技术文档是非常实用的补充资料，可配合本指南一起使用：

1. 版本说明和已知问题：https://xilinx.github.io/Vitis-AI/3.5/html/docs/reference/release_notes.html
2. 《Vitis AI Library 用户指南》(UG1354)
3. 《适用于 Zynq UltraScale+ MPSoC 的 DPUCZDX8G 产品指南》(PG338)
4. 《适用于卷积神经网络的 DPUCAHX8H 产品指南》(PG367)
5. 《适用于 Versal 自适应 SoC 的 DPUCVDX8G 产品指南》(PG389)
6. 《适用于 Versal 自适应 SoC 的 DPUCV2DX8G 产品指南》(PG425)
7. 《Vitis 统一软件平台文档：嵌入式软件开发》(UG1400)
8. 《Vitis 统一软件平台文档：应用加速开发》(UG1393)
9. 《PetaLinux 工具文档：参考指南》(UG1144)

# 修订历史

下表列出了本文档的修订历史。

| 章节 | 修订综述 |
|---|---|
| **2023 年 9 月 28 日 3.5 版** | |
| 不适用 | 仅进行编辑更新。 |
| **2023 年 6 月 29 日 3.5 版** | |
| 第 2 章：模型最优化 | 添加章节。 |
| 第 3 章：量化模型 | 添加：<br>• 量化策略配置。<br>• 使用新增数据格式。<br>• 量化模型的推断。<br>• def export_onnx_model(self, output_dir, verbose)-New。<br>• ONNX Runtime 版本 (vai_q_onnx)。 |
| 第 5 章：部署和运行模型 | • 添加 利用 OnBoard 实现可视化。<br>• 添加 WeGO-Torch C++ 类和 API。<br>• 更新 使用 VOE 进行编程。 |
| 整个文档 | 编辑更新和技术更新。 |
| **2023 年 2 月 24 日 3.0 版** | |
| 常规更新 | 更新链接。 |
| **2023 年 1 月 12 日 3.0 版** | |
| 常规更新 | • 完成适用于新版本的技术更新<br>• 编辑更新 |

| 章节 | 修订综述 |
|---|---|
| **2022 年 6 月 15 日 2.5 版** | |
| 常规更新 | · 完成适用于新版本的技术更新<br>· 编辑更新 |
| **2022 年 1 月 20 日 2.0 版** | |
| 常规更新 | · 更新少量技术细节<br>· 更新支持的卡版本<br>· 编辑更新 |
| 深度学习处理单元 | 更新以包含 Versal DPU |
| vitis_quantize.VitisQuantizer.get_qat_model | 更新实参描述 |
| 受支持的运算符和 DPU 限制 | 更新受支持运算符表 |
| vaitrace 用法 | 更新命令行用法文本 |
| **2021 年 12 月 13 日 1.4.1 版** | |
| 第 3 章：量化模型 | 更新 vai_q_tensorflow2 支持的运算和 API 章节 |
| **2021 年 7 月 22 日 1.4 版** | |
| 第 1 章：Vitis AI 概述 | 新增 Versal AI Core 系列：DPUCVDX8G 章节 |
| TensorFlow 2.x 版本 (vai_q_tensorflow2) | 新增 vai_q_tensorflow2 量化感知训练、利用定制层进行量化 和 vai_q_tensorflow2 用法 章节 |
| PyTorch 版本 (vai_q_pytorch) | 更新 vai_q_pytorch QAT |
| 第 5 章：部署和运行模型 | 更新 Apache TVM、Microsoft ONNX Runtime 和 TensorFlow Lite |
| 第 6 章：模型剖析 | 新增 文本汇总<br>更新 vaitrace 用法 |
| **2021 年 2 月 3 日 1.3 版** | |
| 整个文档 | 更新链接 |
| **2020 年 12 月 17 日 1.3 版** | |
| 整个文档 | 次要更改 |
| 深度学习处理单元 | 添加新主题：Alveo U200/U250 卡：DPUCADF8H 和 Versal AI Core 系列：DPUCVDX8G。 |
| TensorFlow 2.x 版本 (vai_q_tensorflow2) | 新增章节 |
| PyTorch 版本 (vai_q_pytorch) | 新增主题：模块部分量化、vai_q_pytorch 快速微调 和 vai_q_pytorch QAT。 |
| 第 4 章：编译模型 | 新增章节：使用基于 XIR 的工具链执行编译。 |
| 将 DPU 集成到定制平台内 | 新增章节。 |
| VART 编程 API | 新增章节：VART API。 |
| **2020 年 7 月 21 日 1.2 版** | |
| 整个文档 | 次要更改 |
| **2020 年 7 月 7 日 1.2 版** | |
| 整个文档 | · 新增 Vitis AI Profiler 主题。<br>· 新增 Vitis AI 统一 API 简介。 |
| DPU 命名 | 新增主题 |
| 入门 | 更新章节 |
| **2020 年 3 月 23 日 1.1 版** | |
| DPUCAHX8H | 新增主题 |

| 章节 | 修订综述 |
| --- | --- |
| 整个文档 | 新增有关 Alveo U50 支持的内容（支持 U50 DPUV3），包括编译器用法和模型部署描述。 |

# 请阅读：重要法律声明

本文档所示信息仅做参考，其中可能包含不准确的技术信息、疏漏和印刷错误。受诸多原因影响，此处所含信息可能发生更改，也可能无法准确呈现，这些原因包括但不限于产品和路线图变更、组件和主板版本更改、新增模型和/或产品发布、不同制造商之间存在的产品差异、软件更改、BIOS 刷新、固件升级等。任何计算机系统均存在安全性漏洞风险，无法彻底阻止也无法缓解这类风险。AMD 没有任何义务来更新或者以任何其他方式纠正或修改这些信息。但 AMD 保留随时修改这些信息和更改文档内容的权利，AMD 没有任何义务将此类修改或更改通知任何人。此处信息“按原样”提供。AMD 对于本文档内容不作任何陈述或保证，并且对于这些信息中可能出现的任何不准确、错误或疏漏问题不承担任何责任。对于有关任何暗含的非侵权、适销性及适合特定用途的保证，AMD 特此声明不承担任何责任。无论在任何情况下，对于任何人因使用此处包含的任何信息而形成的依赖或者引发的任何直接、间接、特殊或其他后果性损害，AMD 概不负责，即使 AMD 已明确获悉存在发生此类损害的可能性也是如此。

**关于与汽车相关用途的免责声明**

如将汽车产品（部件编号中含“XA”字样）用于部署安全气囊或用于影响车辆控制的应用（“安全应用”），除非有符合 ISO 26262 汽车安全标准的安全概念或冗余特性（“安全设计”），否则不在质保范围内。客户应在使用或分销任何包含产品的系统之前为了安全的目的全面地测试此类系统。在未采用安全设计的条件下将产品用于安全应用的所有风险，由客户自行承担，并且仅在适用的法律法规对产品责任另有规定的情况下，适用该等法律法规的规定。

**版权声明**

© Copyright 2019-2023 AMD 公司，版权所有。AMD、AMD 箭头标识、Alveo、Kria、Versal、Vitis、Vivado、Zynq 及其组合均为 Advanced Micro Devices, Inc. 的商标。 “AMBA”、“AMBA Designer”、“Arm”、“ARM1176JZ-S”、“CoreSight”、“Cortex”、“PrimeCell”、“Mali”和“MPCore”为 Arm Limited 在美国和/或其他国家或地区的商标。 “PCI”、“PCIe”和“PCI Express”均为 PCI-SIG 拥有的商标，且经授权使用。 “OpenCL”和“OpenCL”徽标均为 Apple Inc. 的商标，经 Khronos 许可后方能使用。 此出版物中所使用的其他产品名称仅用于标识目的，可能是其各自所属公司的商标。