

Vitis AI ユーザー ガイド

UG1414 (v1.3) 2021 年 2 月 3 日

この資料は表記のバージョンの英語版を翻訳したもので、内容に相違が生じる場合には原文を優先します。資料によっては英語版の更新に対応していないものがあります。日本語版は参考用としてご使用の上、最新情報につきましては、必ず最新英語版をご参照ください。



改訂履歴

次の表に、この文書の改訂履歴を示します。

セクション	内容
2021 年 2 月 3 日、バージョン 1.3	
文書全体	リンクを更新。
2020 年 12 月 17 日、バージョン 1.3	
文書全体	軽微な変更
深層学習プロセッシング ユニット (DPU)	新規トピックを追加: Alveo U200/U250: DPUCADF8H、Alveo U50/U50LV/U280: DPUCAHX8L、および Versal AI コア シリーズ: DPUCVDX8G。
TensorFlow 2.x バージョン (vai_q_tensorflow2)	新規セクションを追加
Pytorch バージョン (vai_q_pytorch)	新規トピックを追加: モジュールの部分的量子化、vai_q_pytorch 高速微調整、および vai_q_pytorch 量子化微調整。
第 5 章: モデルのコンパイル	新規セクションを追加: XIR ベースのツールチェーンによるコンパイル。
第 10 章: カスタム プラットフォームへの DPU の統合	新たに章を追加。
付録 A: Vitis AI プログラミング インターフェイス	新規セクションを追加: VART API。
2020 年 7 月 21 日 バージョン 1.2	
文書全体	軽微な変更
2020 年 7 月 7 日 バージョン 1.2	
文書全体	<ul style="list-style-type: none"> Vitis AI プロファイラーのトピックを追加。 Vitis AI 統合 API の概要を追加。
DPU 名	新たにトピックを追加。
第 2 章: クイック スタート	この章をアップデート。
2020 年 3 月 23 日 バージョン 1.1	
DPUCAHX8H	新たにトピックを追加。
文書全体	Alveo U50 サポート、およびコンパイラの使用法やモデルの運用方法など U50 DPUV3 の使用に関する説明を追加。

目次

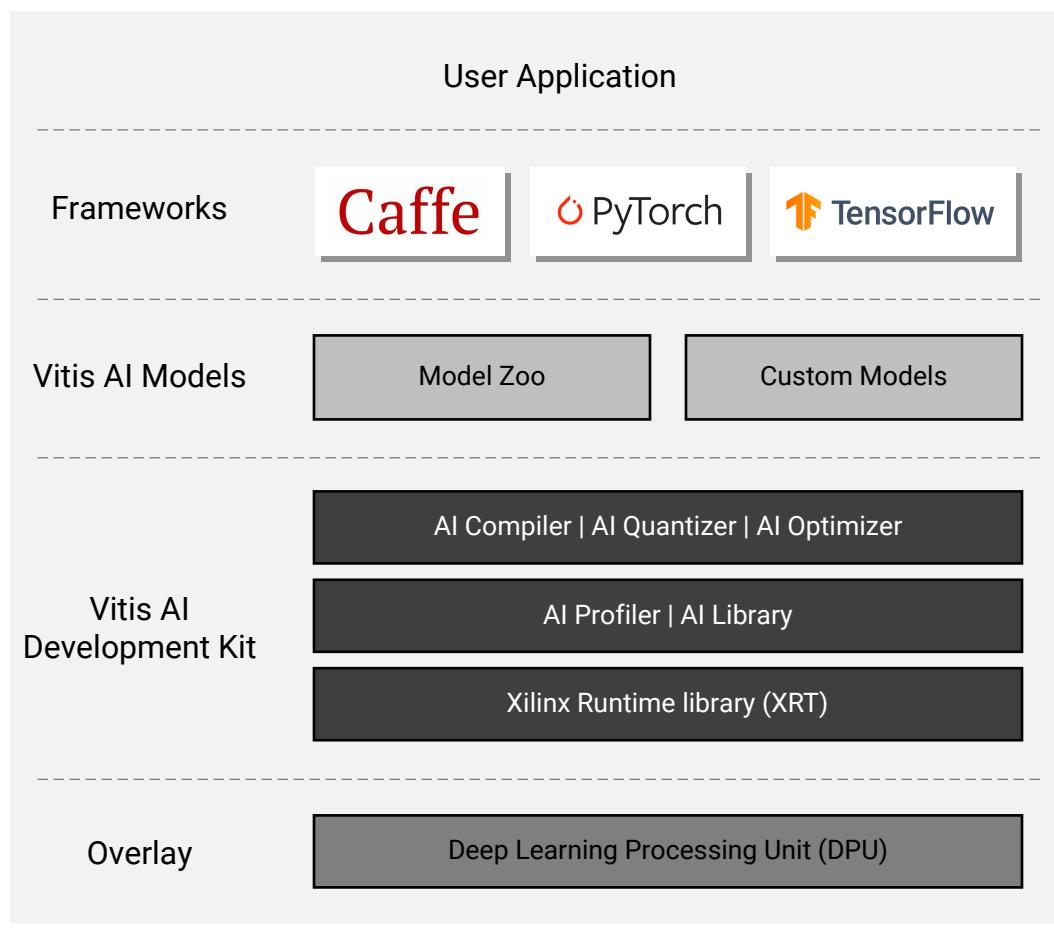
改訂履歴	2
第 1 章: Vitis AI の概要	5
設計プロセス別のコンテンツ ガイド	6
機能	6
Vitis AI ツールの概要	7
Vitis AI コンテナー	20
システム要件	21
開発フローの概要	22
第 2 章: クイック スタート	23
インストールとセットアップ	23
サンプルを実行	33
サポート	38
第 3 章: Vitis AI Model Zoo ネットワークの理解	39
第 4 章: モデルの量子化	40
概要	40
Vitis AI クオンタイザーのフロー	41
TensorFlow 1.x バージョン (vai_q_tensorflow)	42
TensorFlow 2.x バージョン (vai_q_tensorflow2)	54
Pytorch バージョン (vai_q_pytorch)	60
Caffe バージョン (vai_q_caffe)	70
第 5 章: モデルのコンパイル	76
Vitis AI コンパイラ	76
XIR ベースのツールチェーンによるコンパイル	77
DPUCADX8G によるコンパイル	95
VAI_C の使用法	98
第 6 章: モデルの運用と実行	99
Alveo U200/250 上でのモデルの運用と実行	99
VART を使用したプログラミング	99
VART による DPU のデバッグ	101
複数 FPGA のプログラミング	107
Apache TVM および Microsoft ONNX ランタイム	110

第 7 章: モデルのプロファイリング.....	111
Vitis AI プロファイラー	111
第 8 章: モデルの最適化.....	118
第 9 章: ML フレームワークを使用したサブグラフの高速化.....	119
TensorFlow でファンクショナル API 呼び出しを分割.....	119
パーティショナー API.....	119
Caffe のパーティショニング サポート.....	121
第 10 章: カスタム プラットフォームへの DPU の統合.....	123
付録 A: Vitis AI プログラミング インターフェイス	124
VART API.....	124
付録 B: レガシ DNNDK.....	137
DNNDK N2Cube ランタイム.....	137
DNNDK のサンプル.....	139
エッジ向けの DNNDK プログラミング.....	142
DNNDK ユーティリティ	147
DNNDK プロファイラーを使用したプロファイリング.....	153
DNNDK プログラミング API.....	157
付録 C: その他のリソースおよび法的通知.....	210
ザイリンクス リソース.....	210
Documentation Navigator およびデザイン ハブ.....	210
参考資料.....	210
お読みください: 重要な法的通知.....	211

Vitis AI の概要

Vitis™ AI 開発環境は、エッジ デバイスと Alveo™ アクセラレータ カードの両方を含む、ザイリンクス ハードウェア プラットフォーム上での AI 推論を高速化する環境です。これには最適化された IP コア、ツール、ライブラリ、モデル、サンプル デザインが含まれます。Vitis AI は、高い効率性と使いやすさを考えて設計されており、ザイリンクス FPGA および ACAP (Adaptive Compute Acceleration Platform) での AI 推論の高速化を最大限に引き出すことができます。ベースとなる複雑な FPGA や ACAP を抽象化することにより、FPGA の設計経験がないユーザーでも容易に深層学習推論アプリケーションを開発できるようサポートします。

図 1: Vitis AI スタック



設計プロセス別のコンテンツ ガイド

ザイリンクスの資料は、開発タスクに関連する内容を見つけやすいように、標準設計プロセスに基づいて構成されています。Versal™ ACAP デザイン プロセスの [デザイン ハブ](#) は、ザイリンクス ウェブサイトからアクセスできます。この資料では、次の設計プロセスについて説明します。

- 機械学習とデータ サイエンス: Caffe、Pytorch、TensorFlow、またはその他のよく使用されるフレームワークから機械学習モデルを Vitis™ AI にインポートし、その効果を最適化および評価します。この設計プロセスに該当するトピックは、次のとおりです。
 - [第 2 章: クイック スタート](#)
 - [第 4 章: モデルの量子化](#)
 - [第 5 章: モデルのコンパイル](#)
- システム/ソリューション プランニング: システム レベルのコンポーネント、パフォーマンス、I/O、およびデータ転送要件を特定します。ソリューションの PS、PL、および AI エンジンへのアプリケーション マップも含まれます。この設計プロセスに該当するトピックは、次のとおりです。
 - [第 3 章: Vitis AI Model Zoo ネットワークの理解](#)
- エンベデッド ソフトウェア開発: ハードウェア プラットフォームからソフトウェア プラットフォームを作成し、エンベデッド CPU を使用してアプリケーションを開発します。XRT および Graph API も含まれます。この設計プロセスに該当するトピックは、次のとおりです。
 - [第 10 章: カスタム プラットフォームへの DPU の統合](#)
- ホスト ソフトウェア開発: アプリケーション コードおよびアクセラレータを開発します。ライブラリ、XRT、および Graph API の使用も含まれます。この設計プロセスに該当するトピックは、次のとおりです。
 - [第 6 章: モデルの運用と実行](#)
 - [第 9 章: ML フレームワークを使用したサブグラフの高速化](#)
- ハードウェア、IP、プラットフォーム開発: ハードウェア プラットフォーム用の PL IP ブロックの作成、PL カーネルの作成、サブシステムの論理シミュレーション、および Vivado® タイミング、リソース使用、消費電力クロージャの評価を実行します。システム統合用のハードウェア プラットフォームの開発も含まれます。この設計プロセスに該当するトピックは、次のとおりです。
 - [第 10 章: カスタム プラットフォームへの DPU の統合](#)
- システムの統合と検証: システムを統合し、タイミング、リソース使用、消費電力クロージャを含むシステムの機能的なパフォーマンスを検証します。この設計プロセスに該当するトピックは、次のとおりです。
 - [第 7 章: モデルのプロファイリング](#)

機能

Vitis AI には、次の機能があります。

- メインストリーム フレームワークと、さまざまな深層学習タスクを実現する最新モデルをサポート。
- ザイリンクス デバイスですぐに運用できる、最適化済みの包括的なモデル セットを提供。

- モデル量子化、キャリブレーション、微調整をサポートする強力な量子化ツールを提供。アドバンス ユーザー向けに、ザイリンクスはモデルを最大 90% ブルーニングできるオプションの AI オプティマイザーも提供。
- AI プロファイラーではレイヤーごとの分析が可能で、ボトルネックを解消。
- AI ライブラリは、統合された高レベルの C++/Python API を提供し、エッジからクラウドへの移植性を最大限に確保。
- 高効率でスケーラブルな IP コアをカスタマイズして、スループット、レイテンシ、消費電力など、さまざまなアプリケーション要件に対応。

Vitis AI ツールの概要

深層学習プロセッシング ユニット (DPU)

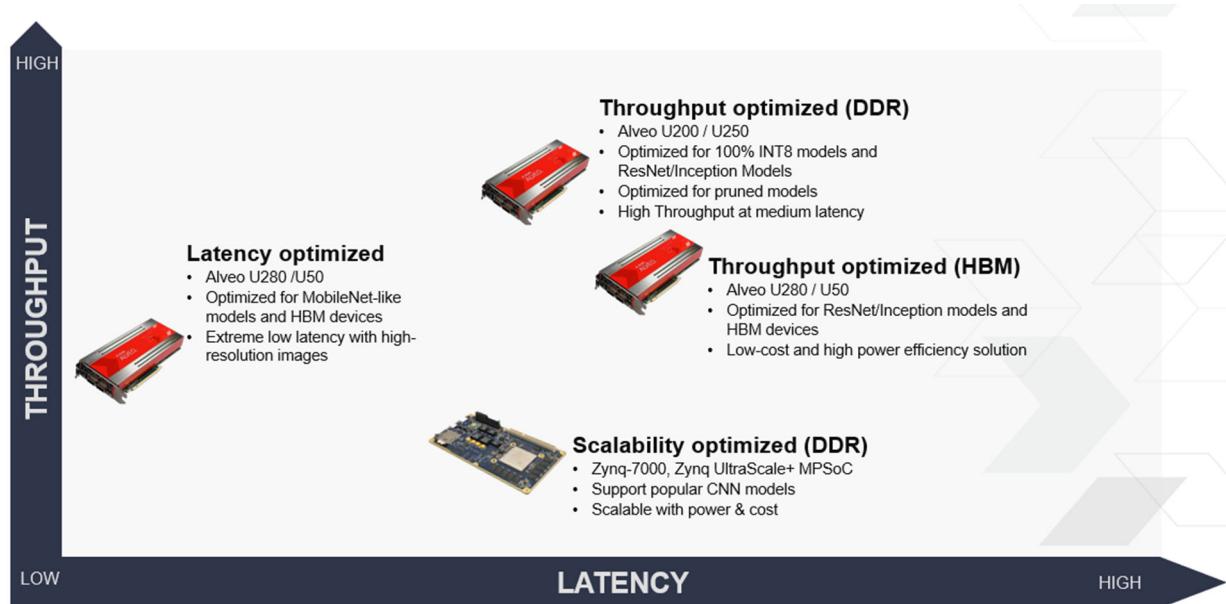
深層学習プロセッシング ユニット (DPU) は、ディープ ニューラル ネットワーク ネットワーク向けに最適化されたプログラマブルなエンジンです。既にハードウェアに実装されたパラメーター指定可能な IP コアであり、配置/配線を実行する必要がありません。DPU は、画像/動画分類、セマンティック セグメンテーション、物体検出/追跡などさまざまなコンピューター ビジョン アプリケーションに広く採用されている深層学習推論アルゴリズムの演算ワークロードを高速に実行できるように設計されています。DPU には、Vitis AI 用の命令セットが含まれているため、深層学習ネットワークを効率的に実装できます。

効率的なテンソル レベルの命令セットは、VGG、ResNet、GoogLeNet、YOLO、SSD、MobileNet など多くの一般的なたたみ込みニューラル ネットワークをサポートしており、これらを高速に実行できるように設計されています。DPU は拡張性に優れているため、エッジからクラウドまでのさまざまなザイリンクス Zynq®-7000 デバイス、Zynq UltraScale+ MPSoC および Alveo ボードに適合可能で、多様なアプリケーション要件を満たすことができます。

コンフィギュレーション ファイル (arch.json) は、Vitis フロー実行時に生成されます。arch.json ファイルは、Vitis AI コンパイラーがモデル コンパイル用に生成します。DPU のコンフィギュレーションを変更した場合、arch.json を新しく生成する必要があります。この新しい arch.json ファイルを使用して、モデルを再度生成します。DPU-TRD では、arch.json ファイルは \$TRD_HOME/prj/Vitis/binary_container_1/link/vivado/vpl/prj/prj.gen/sources_1/bd/xilinx_zcu102_base/ip/xilinx_zcu102_base_DPUCZDX8G_1_0/arch.json に格納されています。

Vitis AI は、ザイリンクス Zynq®-7000 や Zynq® UltraScale+™ MPSoC などのエンベデッド デバイスや Alveo カード (U50、U200、U250、U280) 向けに一連の異なる DPU を提供し、スループット、レイテンシ、スケーラビリティ、消費電力において独自の差別化要素と柔軟性を備えます。

図2: DPUのオプション

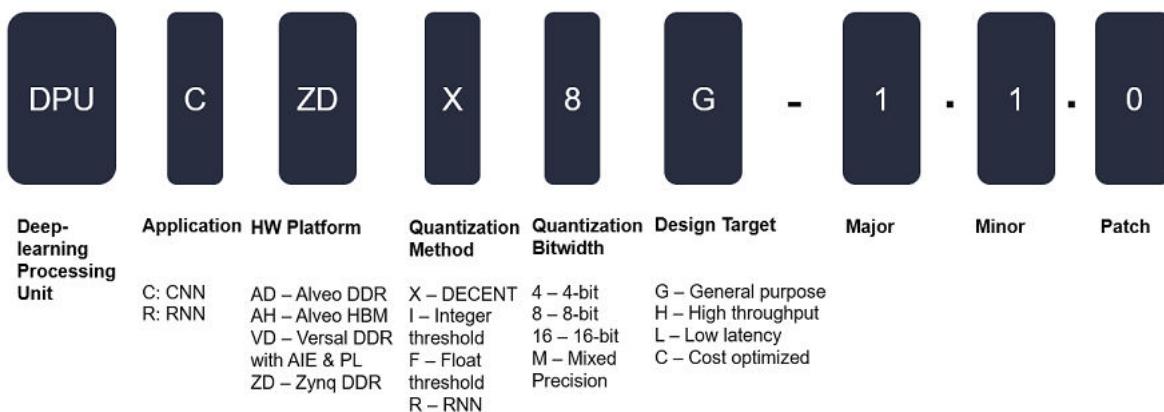


DPU名

Vitis AI 1.2 およびそれ以降のリリースでは、異なる目的に使用される各種の DPU を区別しやすいように、新しい DPU 名を採用しています。DPUv1/v2/v3 の旧名称は廃止されました。

次の図に DPU の新しい命名規則を示します。

図3: DPUの命名規則



DPU名の例

次の表に、旧 DPU 名と新 DPU 名の対応関係と、現在の命名規則を示します。

表1: DPU名の例

例	DPU	アプリケーション	ハードウェア プラットフォーム	量子化の方法	量子化ビット幅	デザイン目標	メジャー	マイナー	パッチ	DPU名
DPUv1	DPU	C	AD	X	8	G	3	0	0	DPUCADX8G-3.0.0
DPUv2	DPU	C	ZD	X	8	G	1	4	1	DPUCZDX8G-1.4.1
DPUv3e	DPU	C	AH	X	8	H	1	0	0	DPUCAHX8H-1.0.0
DPUv3me	DPU	C	AH	X	8	L	1	0	0	DPUCAHX8L-1.0.0
DPUv3int8	DPU	C	AD	F	8	H	1	0	0	DPUCADF8H-1.0.0
XRNN	DPU	R	AH	R	16	L	1	0	0	DPURAHR16L-1.0.0

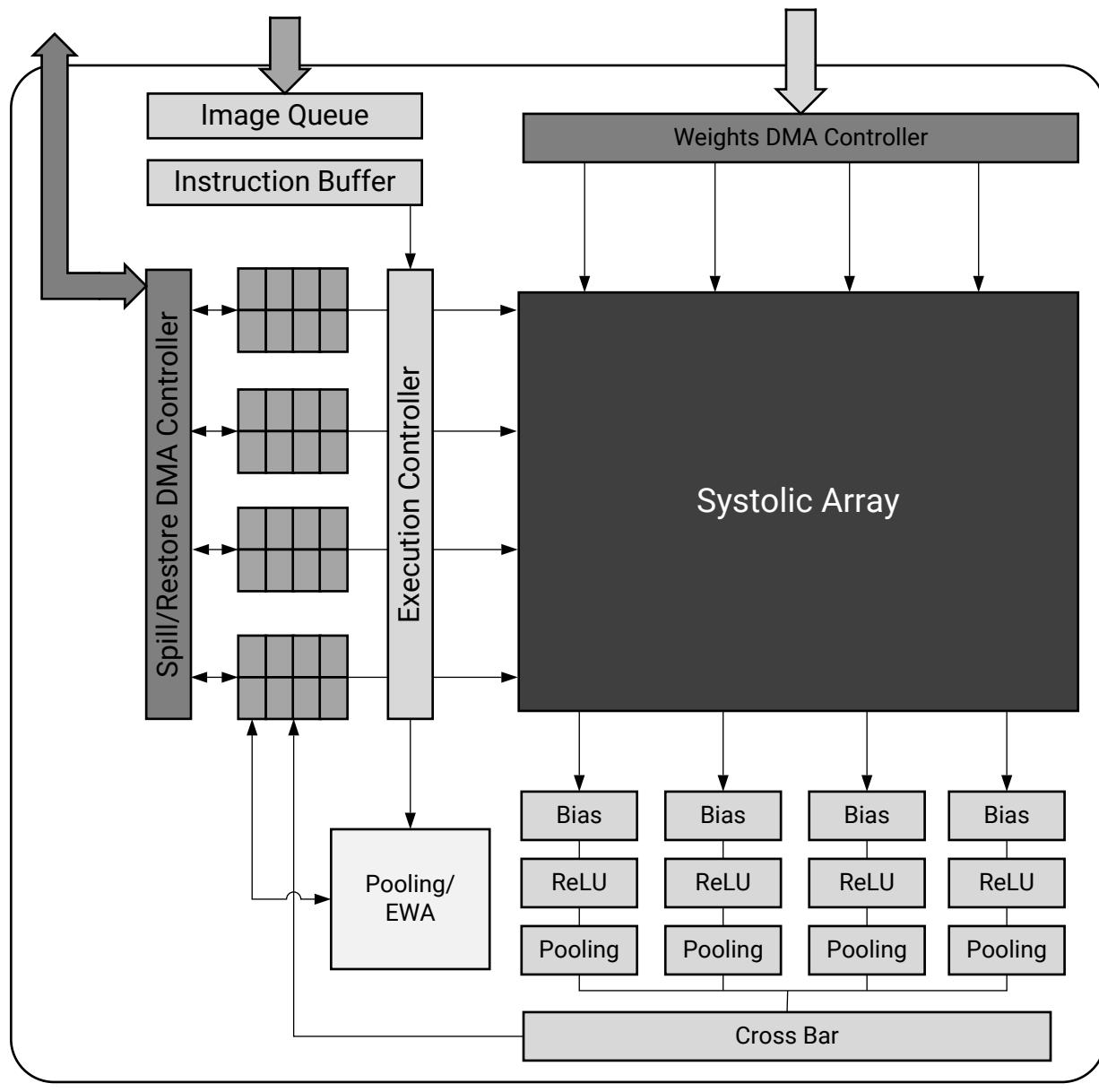
注記:

1. アプリケーション: C - CNN、R - RNN
2. ハードウェア プラットフォーム: AD - Alveo DDR、AH - Alveo HBM、VD - Versal DDR と AI エンジンおよび PL、ZD - Zynq DDR
3. 量子化手法: X - DECENT、F - 浮動小数点しきい値、I - 整数しきい値、R - RNN
4. 量子化帯域幅: 4 - 4 ビット、8 - 8 ビット、16 - 16 ビット、M - 混合精度
5. デザイン目標: G - 汎用、H - 高スループット、L - 低レイテンシ、C - コスト重視

Alveo U200/U250: DPUCADX8G

DPUCADX8G (以前は xDNN) IP コアは、高性能の汎用 CNN プロセッシング エンジン (PE) です。

図4: DPUCADX8G アーキテクチャ



X24609-091620

このエンジンの主な特長は次のとおりです。

- 700MHzで動作する 96x16 DSP シストリック アレイ
- さまざまなカスタムニューラルネットワークグラフを表現するためにシンプルで柔軟性を備えた命令ベースのプログラミングモデル
- UltraRAMで構成される 9 MB オンチップテンソルメモリ
- 分散型オンチップフィルター キャッシュ
- 外部DDRメモリを使用してフィルターおよびテンソルデータを格納
- パイプライン化された Scale、ReLU、および Pooling ブロックで効率が最大化

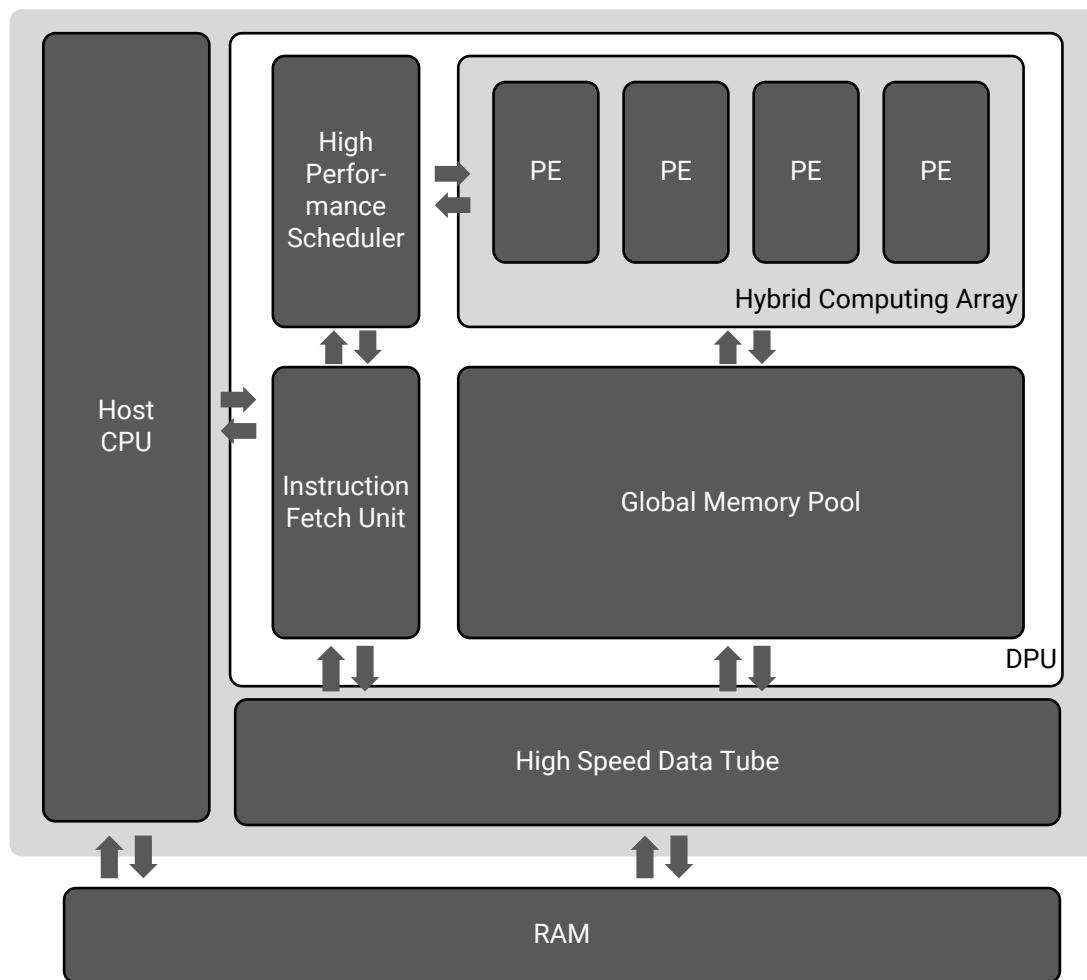
- たたみ込み層を使用する並列処理用のスタンダードアロン Pooling/Eltwise 実行ブロック
- オンチップ テンソル メモリおよびパイプライン化された命令スケジューリングに適合するようにテンソルを分割するためのハードウェア支援型タイリング エンジン
- システムレベルの統合をシンプルにするための標準の AXI-MM および AXI4-Lite 最上位インターフェイス
- オプションのパイプライン化された RGB テンソルたたみ込みエンジンでさらなる効率化

注記: クラウド アプリケーションのスループット向上のため、Vitis AI 1.3 およびそれ以降のリリースでは、Alveo U200/U250 用の新しい DPU (DPUCADF8H) がサポートされます。

Zynq MPSoC: DPUCZDX8G

DPUCZDX8G IP は、ザイリンクス MPSoC デバイス向けに最適化されています。この IP は、選択した Zynq-7000 SoC および Zynq UltraScale+ MPSoC のプログラマブル ロジック (PL) にブロックとして統合して、プロセッシング システム (PS) に直接接続できます。構成可能な DPU IP バージョンは、Vitis AI と同時にリリースされます。DPU は、ユーザーが自由に構成できるようにパラメーター指定が可能なため、PL リソースを最適化したり、有効な機能をカスタマイズできます。カスタマイズされた AI プロジェクトまたは製品に DPU IP を統合する場合は、<https://github.com/Xilinx/Vitis-AI/tree/master/dsa/DPU-TRD> を参照してください。

図 5: DPUCZDX8G アーキテクチャ



X24608-091620

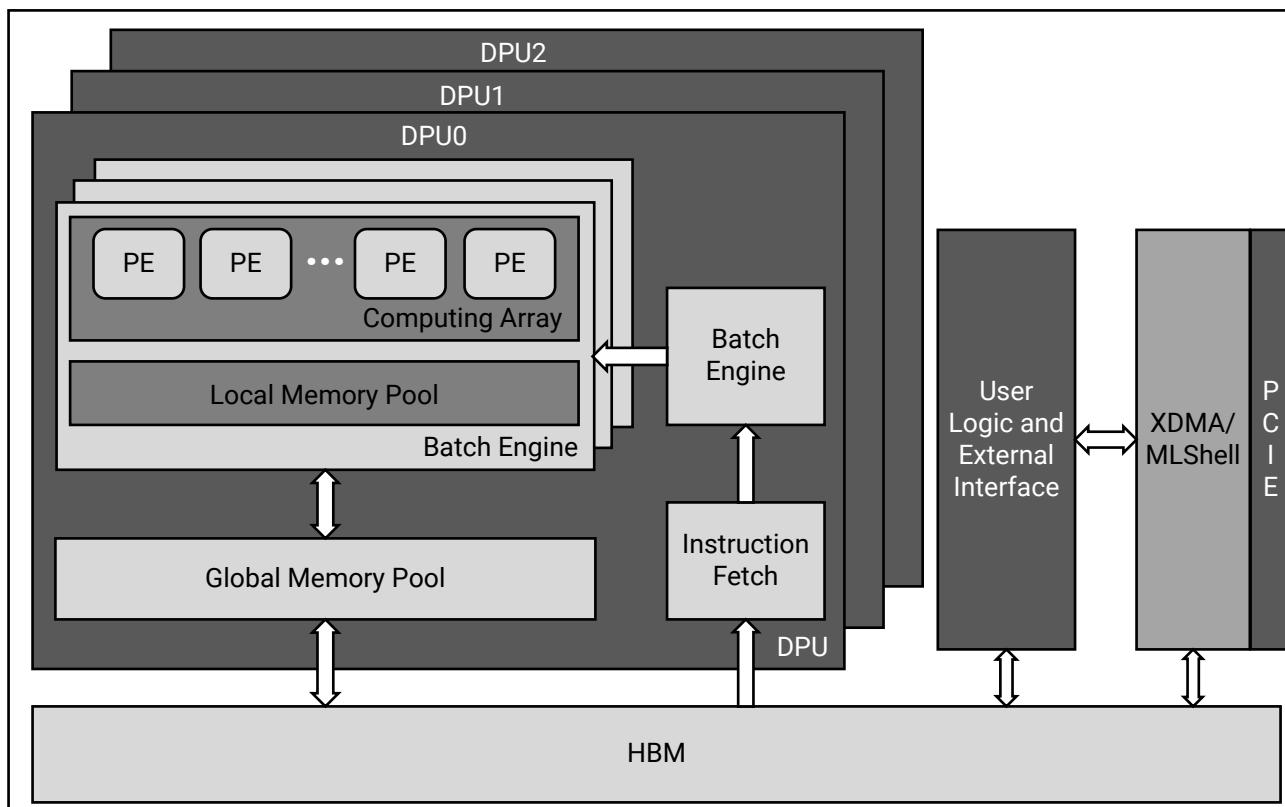
Alveo U50/U280: DPUCAHX8H

ザイリンクス DPUCAHX8H DPU は、主に高スループット アプリケーション向けとして、たたみ込みニューラル ネットワーク (CNN) に最適化されたプログラマブルなエンジンです。このユニットには、高性能スケジューラ モジュール、ハイブリッド コンピューティング アレイ モジュール、命令フェッチ ユニット モジュール、グローバル メモリ プール モジュールがあります。DPU は、多くのたたみ込みニューラル ネットワークを効率よく実行することに特化した命令セットを使用します。運用されているたたみ込みニューラル ネットワークの例としては、VGG、ResNet、GoogLeNet、YOLO、SSD、MobileNet、FPN などがあります。

DPU IP は、一部の Alveo ボードの PL に実装できます。DPU には、ニューラル ネットワークを実行するための命令と、入力イメージおよび一時的なデータと出力データを格納するアクセス可能なメモリ位置が必要です。PL 上で実行されるユーザー定義のユニットにも、適切な設定、命令の挿入、サービスの割り込み、およびデータ転送の調整が必要です。

次の図に、DPU の最上位のブロック図を示します。

図 6: DPUCAHX8H の最上位ブロック図

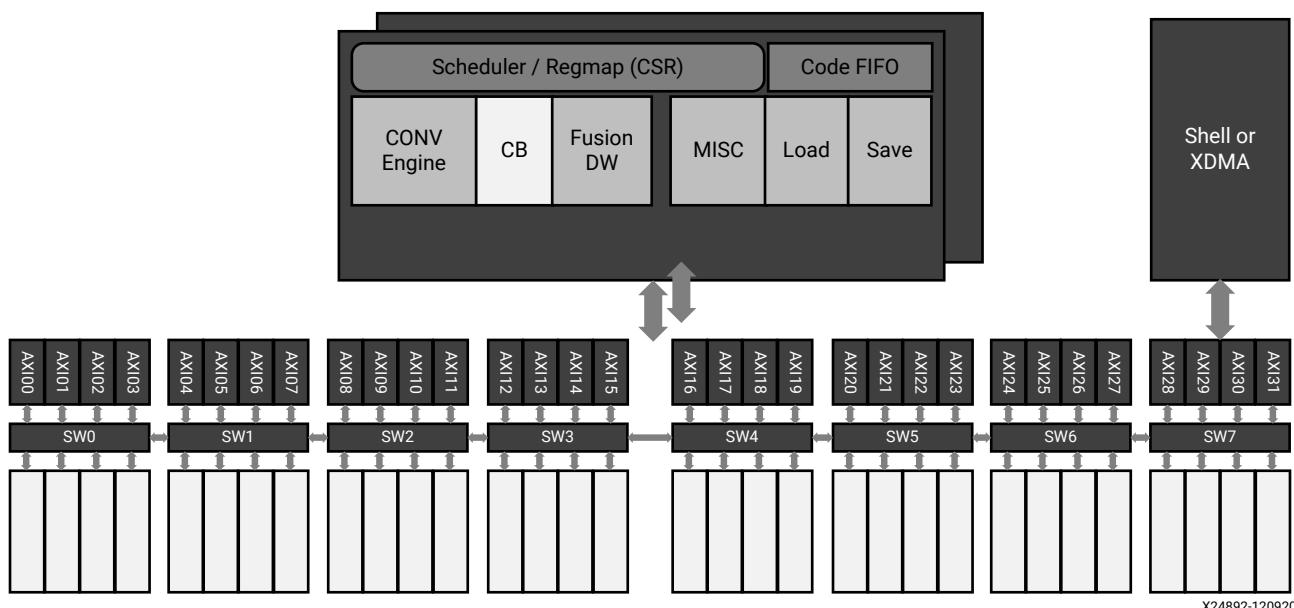


X24606-091620

Alveo U50/U50LV/U280: DPUCAHX8L

DPUCAHX8L IP は、U50/U50LV や U280 といった HBM カード向けに最適化され、低レイテンシ アプリケーション用に設計された、新しい汎用 CNN アクセラレータです。この DPU は、新しい低レイテンシの DPU マイクロ アーキテクチャを採用し、4TOPs ~ 5.3TOPs の MAC アレイをサポートする HBM メモリ サブシステムを備えています。また、隣接するたたみ込みエンジンおよび depthwise たたみ込みエンジンをサポートし、計算処理の並列性を向上させています。さらに、階層構造のメモリ システム (URAM および HBM) をサポートし、データ転送能力を最大限に高めています。この低レイテンシの DPU IP により、Xcompiler は、スーパー レイヤー インターフェイスと、カーネル融合およびグラフ分割など多数の新しいコンパイル ストラテジをサポートします。

図 7: DPUCAHX8L のアーキテクチャ



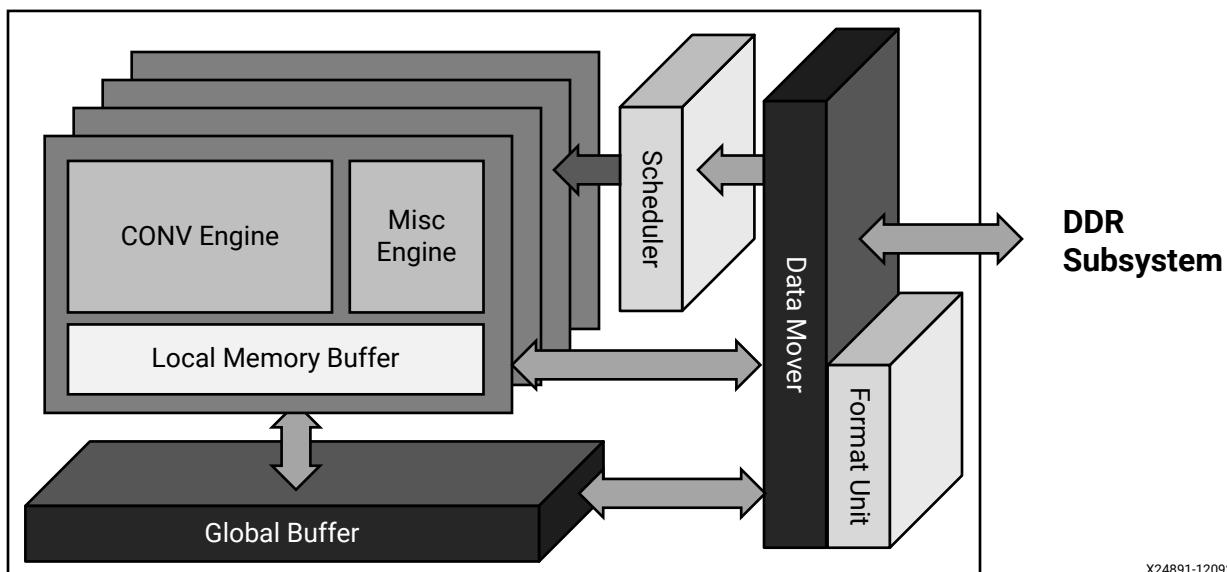
Alveo U200/U250: DPUCADF8H

DPUCADF8H は、Alveo U200/U250 向けに最適化された DPU であり、高スループット アプリケーションをターゲットとしています。DPUCADF8H の主な機能は次のとおりです。

- スループット重視の高効率コンピューティング エンジン: 各種ワーカロードでスループットが 1.5 ~ 2.0 倍に向 上
- たたみ込みニューラル ネットワークを幅広くサポート
- 圧縮されたたたみ込みニューラル ネットワークに最適
- 高分解能画像に最適化

次の図に、最上位のブロック図を示します。

図 8: DPUCADF8H のアーキテクチャ



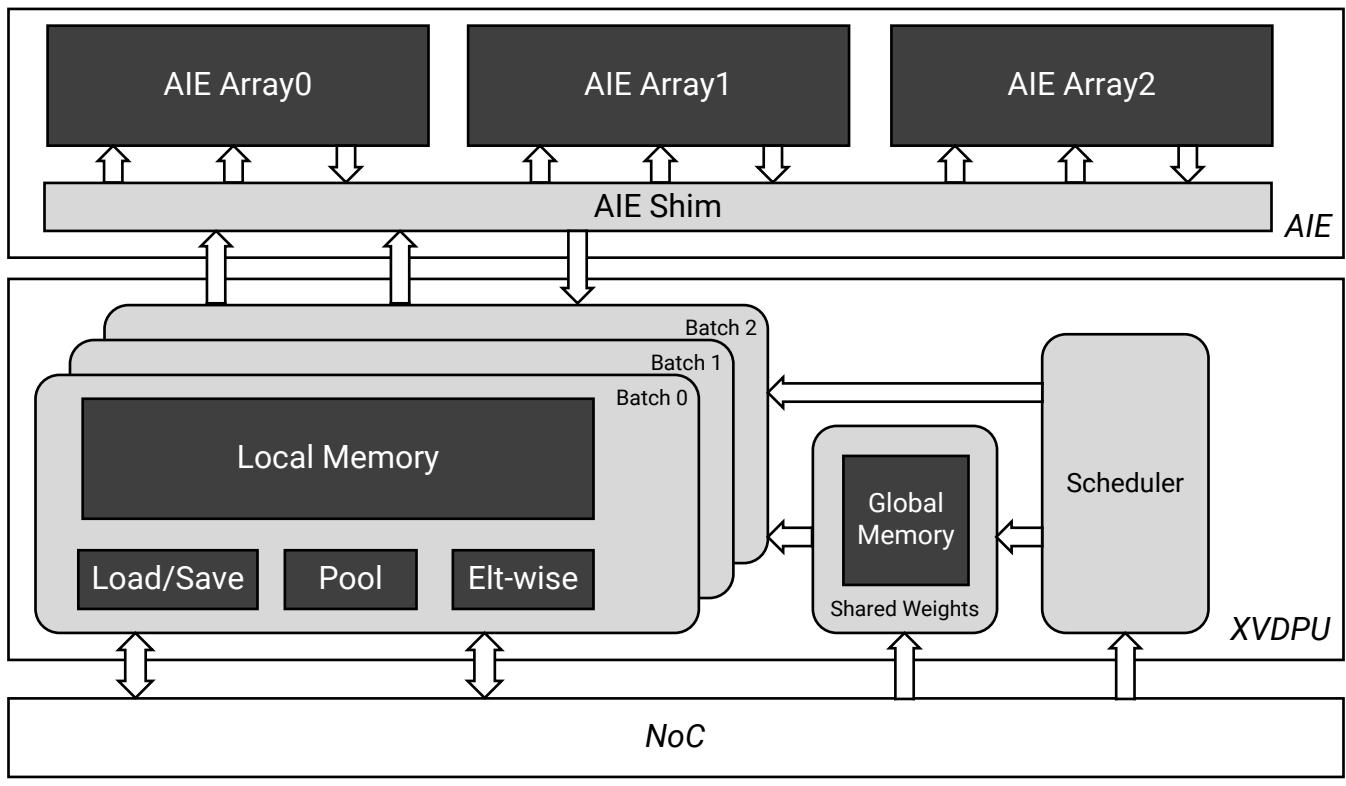
X24891-120920

Versal AI コア シリーズ: DPUCVDX8G

DPUCVDX8G は、Versal AI コア シリーズ向けに最適化された高性能な汎用 CNN プロセッシング エンジンです。Versal デバイスは、従来の FPGA、CPU、および GPU と比べて、より優れた単位ワットあたりの性能を提供します。DPUCVDX8G は、AI エンジンと PL で構成されます。この IP は、ユーザーが自由に構成できるようにパラメーター指定が可能なため、AI エンジンと PL リソースを最適化したり、有効な機能をカスタマイズできます。

次の図に、DPUCVDX8G の最上位のブロック図を示します。

図 9: DPUCVDX8G のアーキテクチャ

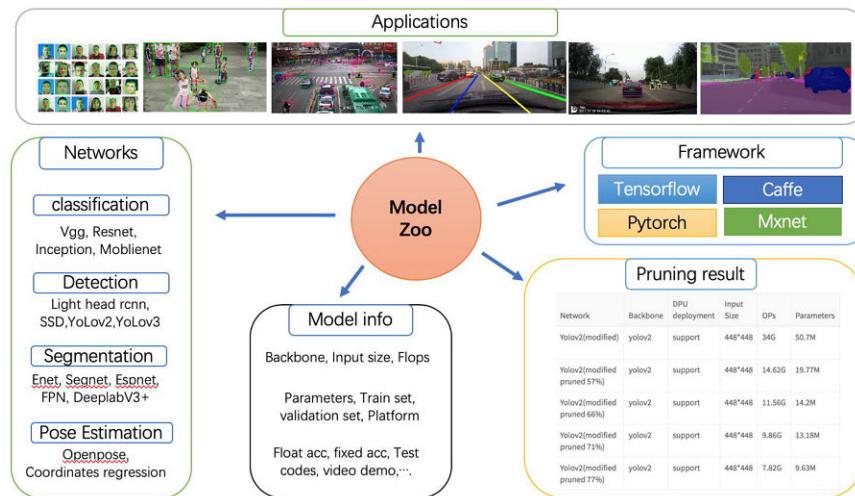


AI Model Zoo

AI Model Zoo には、ザイリンクス プラットフォームで深層学習推論をすばやく運用するための、最適化済み深層学習モデルが含まれています。これらのモデルは、ADAS/AD、ビデオ監視、ロボティクス、データセンターなどのさまざまなアプリケーションを対象としています。これらのトレーニング済みモデルから始めることで、深層学習アクセラレーションのメリットを享受できます。

詳細は、GitHub で [Vitis AI Model Zoo](#) を参照してください。

図 10: AI Model Zoo

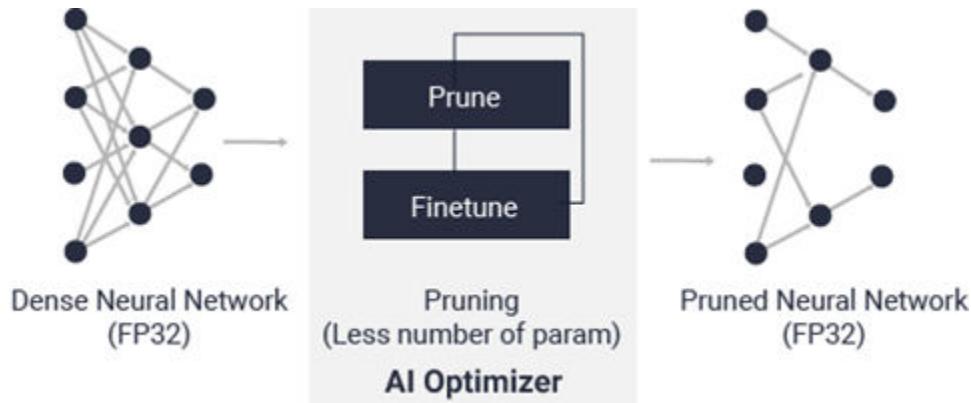


AI オプティマイザー

世界最先端のモデル圧縮技術により、精度の低下を最小限に抑えながら、複雑なモデルを 1/5 から最大 1/50 までに圧縮できます。ディープ圧縮で次世代レベルの AI 推論性能が実現します。AI オプティマイザーの詳細は、『Vitis AI オプティマイザー ユーザー ガイド』(UG1333) を参照してください。

AI オプティマイザーを実行するには、商用ライセンスが必要です。詳細は、ザイリンクス販売代理店までお問い合わせください。

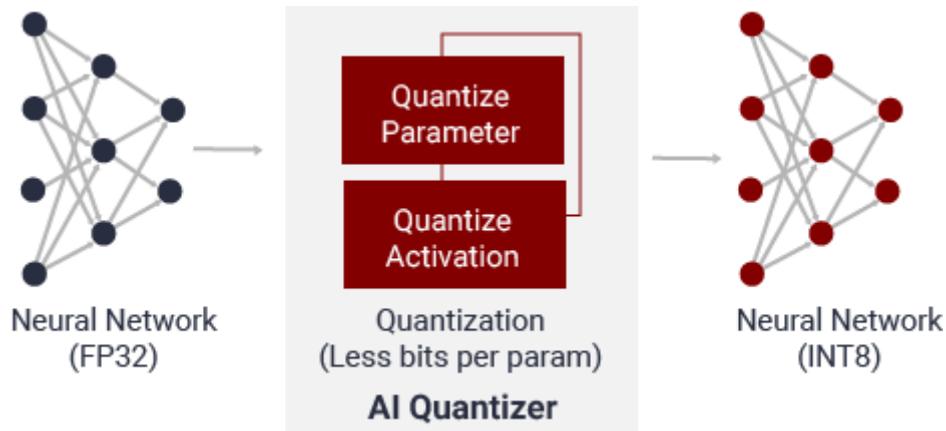
図 11: AI オプティマイザー



AI クオンタイザー

AI クオンタイザーは、32 ビット浮動小数点の重みやアクティベーション コードを INT8 などの固定小数点に変換することで、予測精度を損なうことなく計算の複雑レベルを軽減できます。固定小数点ネットワーク モデルの方が、浮動小数点モデルよりも必要なメモリ帯域幅が狭く、速度と電力効率が向上します。

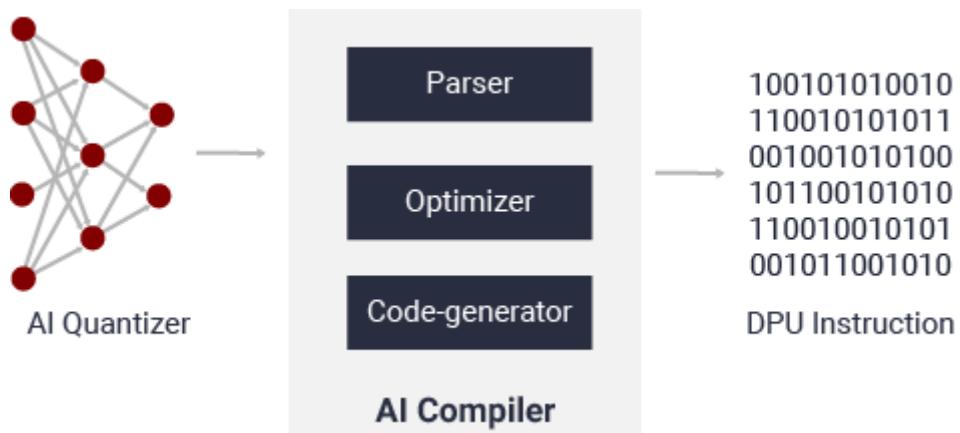
図 12: AI クォンタイザー



AI コンパイラー

AI コンパイラは、AI モデルを効率のよい命令セットとデータフロー モデルにマップします。また、レイヤーの融合や命令スケジューリングなどの高度な最適化を実行し、オンチップ メモリを可能な限り再利用します。

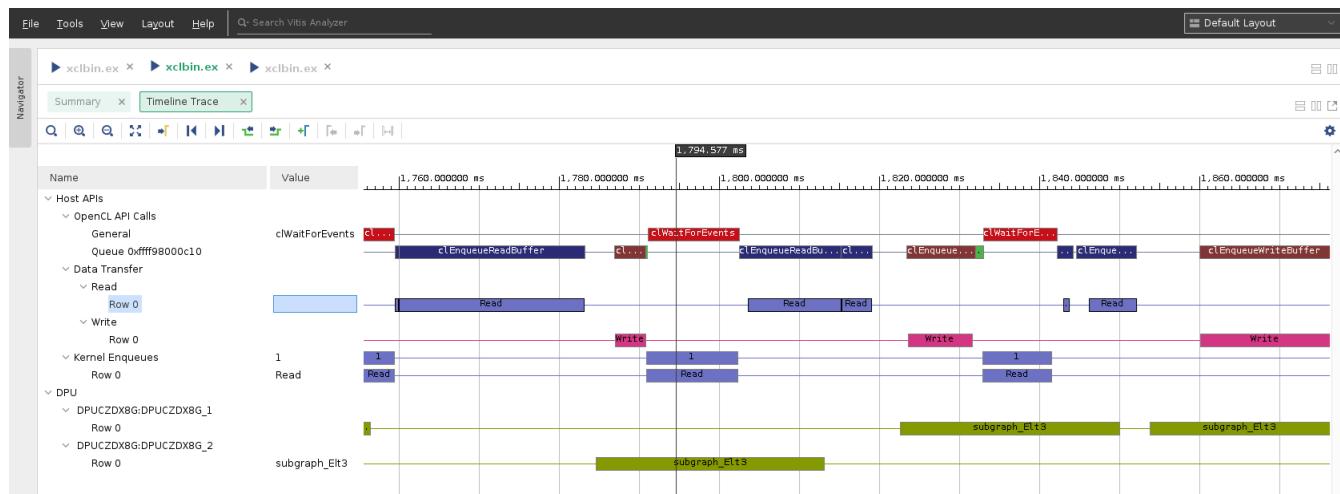
図 13: AI コンパイラー



AI プロファイラー

Vitis AI プロファイラーは、AI アプリケーションのプロファイリングと視覚化を実行し、ボトルネックの認識や多様なデバイス間における演算リソースの割り当てに役立ちます。使用は簡単で、コードの変更は必要ありません。関数呼び出しや実行時間のトレースが可能であり、CPU、DPU、メモリ使用率などのハードウェア情報も収集できます。

図 14: AI プロファイラー

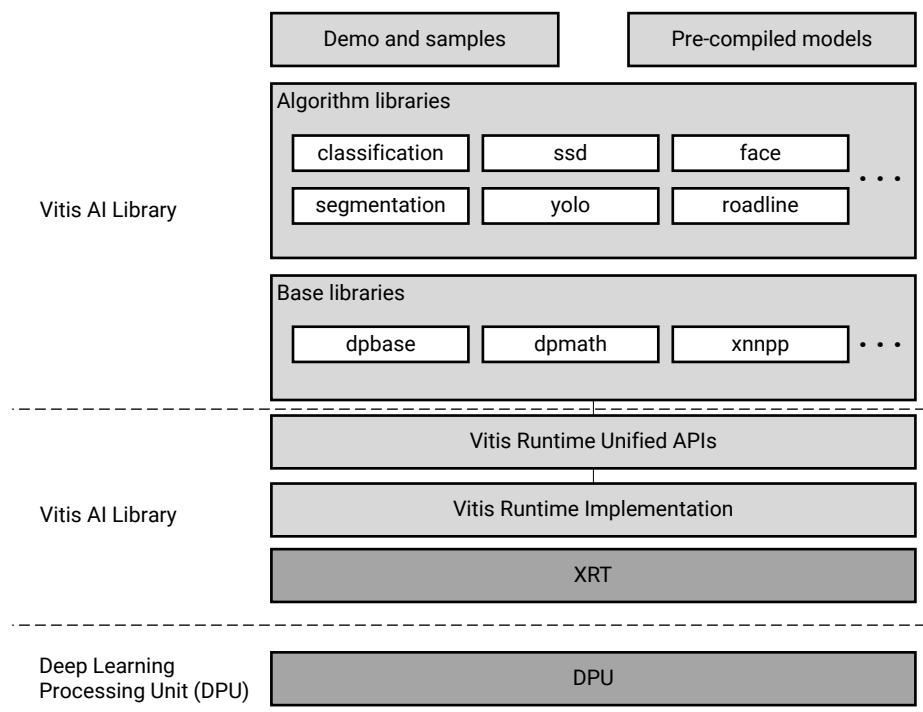


AI ライブラリ

Vitis AI ライブラリは、DPU を使用して効率的な AI 推論を実現するために構築された高レベルのライブラリと API です。XRT を完全にサポートし、Vitis ランタイム統合 API を備えた Vitis AI ランタイムをベースに構築されています。

Vitis AI ライブラリは、効率的かつ高品質のニューラル ネットワークをカプセル化することで、統合された使いやすいインターフェイスを提供します。これにより、深層学習や FPGA の知識がないユーザーでも、深層学習ニューラルネットワークを簡単に使用できます。Vitis AI ライブラリを使用することで、基礎をなすハードウェアに時間を費やすことなく、アプリケーション開発により重点を置くことができます。

図 15: AI ライブラリ



AI ランタイム

AI ランタイムにより、アプリケーションはクラウドとエッジの両方に対応する高レベルの統合されたランタイム API を使用でき、クラウドとエッジ間の運用がシームレスかつ効率的になります。

AI ランタイム API の機能は次のとおりです。

- アクセラレータへジョブを非同期送信
- アクセラレータからジョブを非同期取得
- C++ および Python で実装
- マルチスレッドおよびマルチプロセスの実行に対応

クラウドの場合

クラウド アクセラレータには複数の独立した演算ユニット (CU) があり、それぞれが異なる AI モデルで動作するよう にプログラムしたり、同じ AI モデルで動作するようにプログラムして最大スループットを達成することが可能です。

クラウド ランタイムでは、新しい AI リソース マネージャーを使用することで、複数の FPGA リソースによるアプリケーションのスケーリングが容易になります。アプリケーションは、使用する特定の FPGA カードを指定する必要がなくなりました。アプリケーションは単一の CU または単一の FPGA を要求でき、AI リソース マネージャーが、要求を満たすような使用されていないリソースを返します。AI リソース マネージャーは Docker コンテナーを使用して管理します。また、同じホスト上の複数ユーザー/テナントにも対応します。

Vitis AI ランタイム

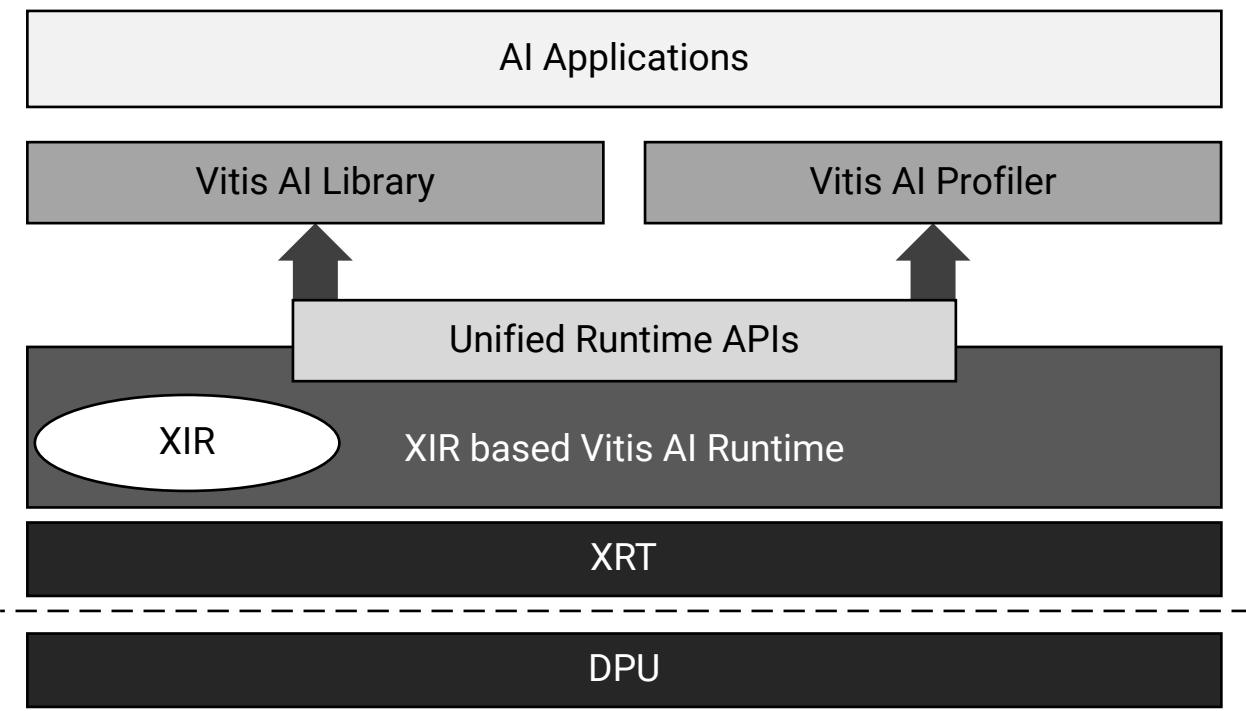
Vitis AI ランタイム (VART) は、DPUCZDX8G、DPUCADX8G、DPUCADF8H、および DPUCAHX8H をベースとする デバイスに最適な次世代ランタイムです。DPUCZDX8G および DPUCADF8H は ZCU102 や ZCU104 などのエッジ デバイスに使用され、DPUCADX8G は Alveo U200 や U250 などのエッジ デバイスに使用され、DPUCAHX8H は Alveo U50、U50LV、および U280 などのクラウド デバイスに使用されます。DPUCVDX8G は VCK190 などの Versal 評価ボードに使用されます。次の図に VART のフレームワークを示します。Vitis AI リリースでは、VART は XRT を ベースにしています。

現在、Vitis AI には次の 2 種類のランタイムが付属しています。

- VART: ザイリンクス中間表現 (XIR) をベースにしています。これはグラフ ベースの中間表現で、Vitis AI の公式データ交換規格です。
- n2cube: レガシ ディープ ニューラル ネットワーク開発キット (DNNDK) に含まれ、互換性維持のために提供されています。

注記: DNNDK は、Vitis AI 1.4 およびそれ以降のリリースではサポートされません。

図 16: VART スタック



X24605-091620

Vitis AI コンテナー

Vitis AI 1.3 リリースでは、コンテナー技術を使用して AI ソフトウェアを割り当てます。このリリースには、次のコンポーネントが含まれています。

- ツール コンテナー
- Zynq UltraScale+ MPSoC 向けランタイム パッケージ
- GitHub で公開されているサンプル (<https://github.com/Xilinx/Vitis-AI>)
- Vitis AI Model Zoo (<https://github.com/Xilinx/Vitis-AI/tree/master/models/AI-Model-Zoo>)

ツール コンテナー

ツール コンテナーの構成は次のとおりです。

- Docker Hub で配布されるコンテナー: <https://hub.docker.com/r/xilinx/vitis-ai/tags>
- 統合コンパイラ フローには次のフローが含まれます。
 - DPUCZDX8G 向けのコンパイラ フロー (エンベデッド)
 - DPUCAHX8H 向けのコンパイラ フロー (クラウド)
 - DPUCAHX8L 向けのコンパイラ フロー (クラウド)
 - DPUCVDX8G 向けのコンパイラ フロー (クラウド)

- DPUCVDX8H 向けのコンパイラ フロー (クラウド)
- DPUCADX8G 向けのコンパイラ フロー (クラウド)
- DPUCADF8H 向けのコンパイラ フロー (クラウド)
- 構築済みの Conda 環境でフレームワークを実行する。
 - Conda で Caffe ベース フロー用の vitis-ai-caffe をアクティベート
 - Conda で TensorFlow ベース フロー用の vitis-ai-tensorflow をアクティベート
 - Conda で TensorFlow2 ベース フロー用の vitis-ai-tensorflow2 をアクティベート
 - Conda で PyTorch ベース フロー用の vitis-ai-pytorch をアクティベート
- Alveo ランタイム ツール

MPSOC デバイス向けランタイム パッケージ

- コンテナー パス URL: https://japan.xilinx.com/bin/public/openDownload?filename=vitis_ai_2020.2-r1.3.0.tar.gz
- 内容
 - PetaLinux SDK およびクロス コンパイラ ツール チェーン
 - 2020.2 リリースに基づく Vitis AI ボード パッケージ (Vitis AI の新世代ランタイム VART を含む)
- モデルおよびオーバーレイ bin ファイル (<https://github.com/Xilinx/Vitis-AI>)
 - すべての公開されているトレーニング済みモデル
 - すべての Zynq UltraScale+ MPSOC および Alveo アクセラレータ カード オーバーレイ
 - モデルおよびオーバーレイのダウンロードとインストールを自動化するスクリプトが含まれる。

システム要件

次の表に、コンテナーと Alveo ボードを実行する際のシステム要件を示します。

表 2: システム要件

コンポーネント	要件
FPGA	ザイリンクス Alveo U50、U50LV、U200、U250、U280、ザイリンクス ZCU102、ZCU104、VCK190
マザーボード	PCI Express® 3.0 準拠 (1 デュアル幅 × 16 スロット)
システム電源	225W
オペレーティング システム	<ul style="list-style-type: none">◦ Linux、64 ビット◦ Ubuntu 16.04、18.04◦ CentOS 7.4、7.5◦ RHEL 7.4、7.5
GPU (量子化を高速化するオプション)	NVIDIA GPU は、NVIDIA P100、V100 と同様に CUDA 9.0 以上をサポート
CUDA ドライバー (量子化を高速化するオプション)	CUDA のバージョンと互換性のあるドライバー: CUDA 9.0 には NVIDIA-384 以上、CUDA 10.0 には NVIDIA-410 以上

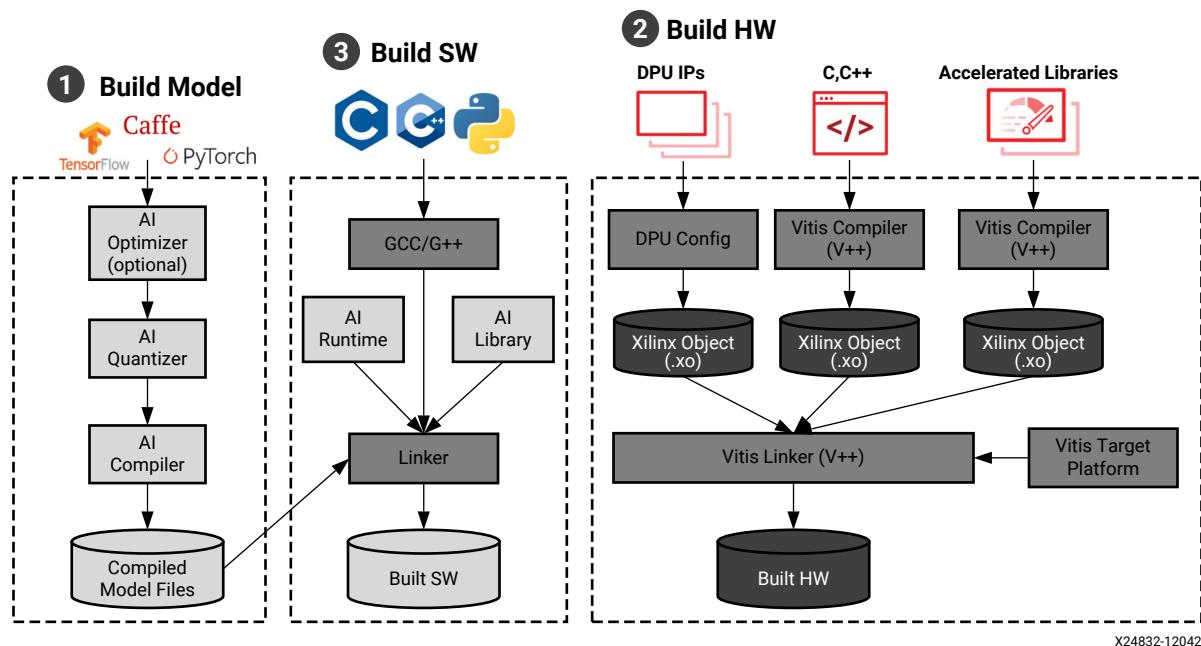
表2: システム要件(続き)

コンポーネント	要件
Docker バージョン	19.03 以上

開発フローの概要

Vitis™ AI の推奨開発フローを次の図に示します。このフローは 3 つの基本的なステップで構成され、Vitis AI と Vitis IDE が必要です。

図17: Vitis AI フロー



- ホストマシンの Vitis AI ツールチェーンを使用して、モデルを構築します。このツールチェーンは、トレーニング済みの浮動小数点モデルを入力として使用し、AI オプティマイザーでそれらのモデルを実行します(オプション)。
- カスタムハードウェア プラットフォームは、Vitis ターゲット プラットフォームをベースとする Vitis ソフトウェア プラットフォームを使用して構築されます。生成されるハードウェアには、DPU IP およびその他のカーネルが含まれます。Vitis AI リリース パッケージには、事前に構築された SD カード イメージ (ZCU102/104 用) と Alveo™ シェルが含まれており、それらを使用して設計を開始してアプリケーションを開発できます。Vivado® Design Suite を使用して DPU を統合し、要件に適合するカスタム ハードウェアを構築することもできます。詳細は、[第10章: カスタム プラットフォームへの DPU の統合](#) を参照してください。
- 構築されたハードウェアで実行される、実行可能なソフトウェアを構築できます。C++ または Python で作成したアプリケーションで Vitis AI ランタイムと Vitis AI ライブラリを呼び出し、コンパイルされたモデル ファイルをロードして実行できます。

クイック スタート

インストールとセットアップ

Vitis AI 開発キットのダウンロード

Vitis™ AI ソフトウェアは、Docker Hub から入手できます。Vitis AI には次の 2 つのパッケージがあります。

- Vitis AI ツール Docker: [xilinx/vitis-ai:latest](#)
- エッジ向け Vitis AI ランタイム パッケージ

ツール コンテナーには、クラウド DPU 用の Vitis AI クオンタイザー、AI コンパイラ、および AI ランタイムが含まれています。エッジ用の Vitis AI ランタイム パッケージはエッジ DPU 開発用であり、ザイリンクスの ZCU102 および ZCU104 評価ボード用の Vitis AI ランタイム インストール パッケージ、および Arm® GCC クロス コンパイル ツール チェーンが含まれています。

Vitis AI 開発キット v1.3 リリースでサポートされるザイリンクスの FPGA デバイスおよび評価ボードは次のとおりです。

- クラウド: Alveo™ カード U200、U250、U280、U50、U50LV、および Versal ACAP 評価ボード VCK190。
- エッジ: Zynq® UltraScale+™ MPSoC 評価ボード ZCU102、ZCU104。

ホストのセットアップ

Vitis AI ツールおよびリソースを含むコンテナーのインストールには、次の 2 つのオプションがあります。

1. Docker Hub 上のあらかじめ構築されたコンテナー: [xilinx/vitis-ai](#)
2. Docker レシピを使用してコンテナーをローカルで構築する: [Docker レシピ](#)

インストールの手順は次のとおりです。

1. [Docker をインストールします](#) (Docker がマシンにインストールされていない場合)。
2. [Linux でのインストール後の手順](#) に従って、Linux ユーザーがグループ Docker に所属していることを確認します。
3. Vitis AI リポジトリをコピーして、サンプル、リファレンス コード、およびスクリプトを取得します。

```
git clone --recurse-submodules https://github.com/Xilinx/Vitis-AI  
cd Vitis-AI
```

4. [Docker コンテナーを実行します](#)。

- Docker Hub から CPU イメージを実行します。

```
docker pull xilinx/vitis-ai:latest
./docker_run.sh xilinx/vitis-ai
```

- CPU イメージをローカルで構築し、実行します。

```
cd docker
./docker_build_cpu.sh

# After build finished
cd ..
./docker_run.sh xilinx/vitis-ai-cpu:latest
```

- GPU イメージをローカルで構築し、実行します。

```
cd docker
./docker_build_gpu.sh

# After build finished
cd ..
./docker_run.sh xilinx/vitis-ai-gpu:latest
```

ホストのセットアップ (VART を使用)

エッジの場合 (DPUCZDX8G)

次の手順に従って、エッジ用のホストをセットアップします。

1. [こちら](#) から sdk-2020.2.0.0.sh をダウンロードします。
2. クロスコンパイルシステム環境をインストールします。

```
./sdk-2020.2.0.0.sh
```

3. プロンプトに従ってインストールします。次の図に、インストールプロセスを示します。

推奨: インストールパスには ~/petalinux_sdk を推奨します。どのパスを選択した場合も、インストールパスには書き込みパーミッションが必要です。ここでは、~/petalinux_sdk にインストールします。

4. インストールが完了したら、プロンプトに従って次のコマンドを入力します。

```
source ~/petalinux_sdk/environment-setup-aarch64-xilinx-linux
```

注記: 現在のターミナルを閉じた場合は、次に新規ターミナルを開いたときに上記の手順をもう一度実行して環境を設定する必要があります。

5. [こちら](#) から vitis_ai_2020.2-r1.3.0.tar.gz をダウンロードし、PetaLinux システムにインストールします。

```
tar -xzvf vitis_ai_2020.2-r1.3.0.tar.gz -C ~/petalinux_sdk/sysroots/aarch64-xilinx-linux
```

6. サンプルをクロスコンパイルします(resnet50 を例として使用)。

```
cd Vitis-AI/demo/VART/resnet50
bash -x build.sh
```

コンパイルプロセスでエラーが報告されず、実行可能ファイル resnet50 が生成されている場合は、ホスト環境が正しくインストールされています。

クラウドの場合 (DPUCAHX8H)

次の手順に従って、クラウド用のホストをセットアップします。これらの手順は、U50、U50LV、および U280 カードに適用されます。

1. Docker コンテナーを起動します。Docker イメージがロードされて実行されると、Vitis AI ランタイムが Docker システムに自動的にインストールされます。
2. こちらから xclbin ファイルをダウンロードします。ファイルを解凍し、Alveo カードを選択してインストールします。ここでは、U50 の例を示します。

```
tar -xzvf alveo_xclbin-1.3.0.tar.gz  
cd alveo_xclbin-1.3.0/U50/6E300M  
sudo cp dpu.xclbin hbm_address_assignment.txt /usr/lib
```

DPUCAHX8L については、U50LV を例として使用します。

```
tar -xzvf alveo_xclbin-1.3.0.tar.gz  
cd alveo_xclbin-1.3.0/U50LV-V3ME/1E250M  
sudo cp dpu.xclbin /opt/xilinx/overlaybins/  
export XLNX_VART_FIRMWARE=/opt/xilinx/overlaybins/dpu.xclbin
```

注記: サーバーにインストールするカードが 2 つ以上あり、プログラムを実行するカードを指定する場合は、XLNX_ENABLE_DEVICES を設定します。次のオプションがあります。

- デバイス 0 を DPU に使用する場合は、`export XLNX_ENABLE_DEVICES=0` に設定します。
- デバイス 0、デバイス 1、およびデバイス 2 を DPU に使用する場合は、`export XLNX_ENABLE_DEVICES=0,1,2` に設定します。
- この環境変数を設定しない場合、デフォルトではすべての利用可能なデバイスが DPU に使用されます。

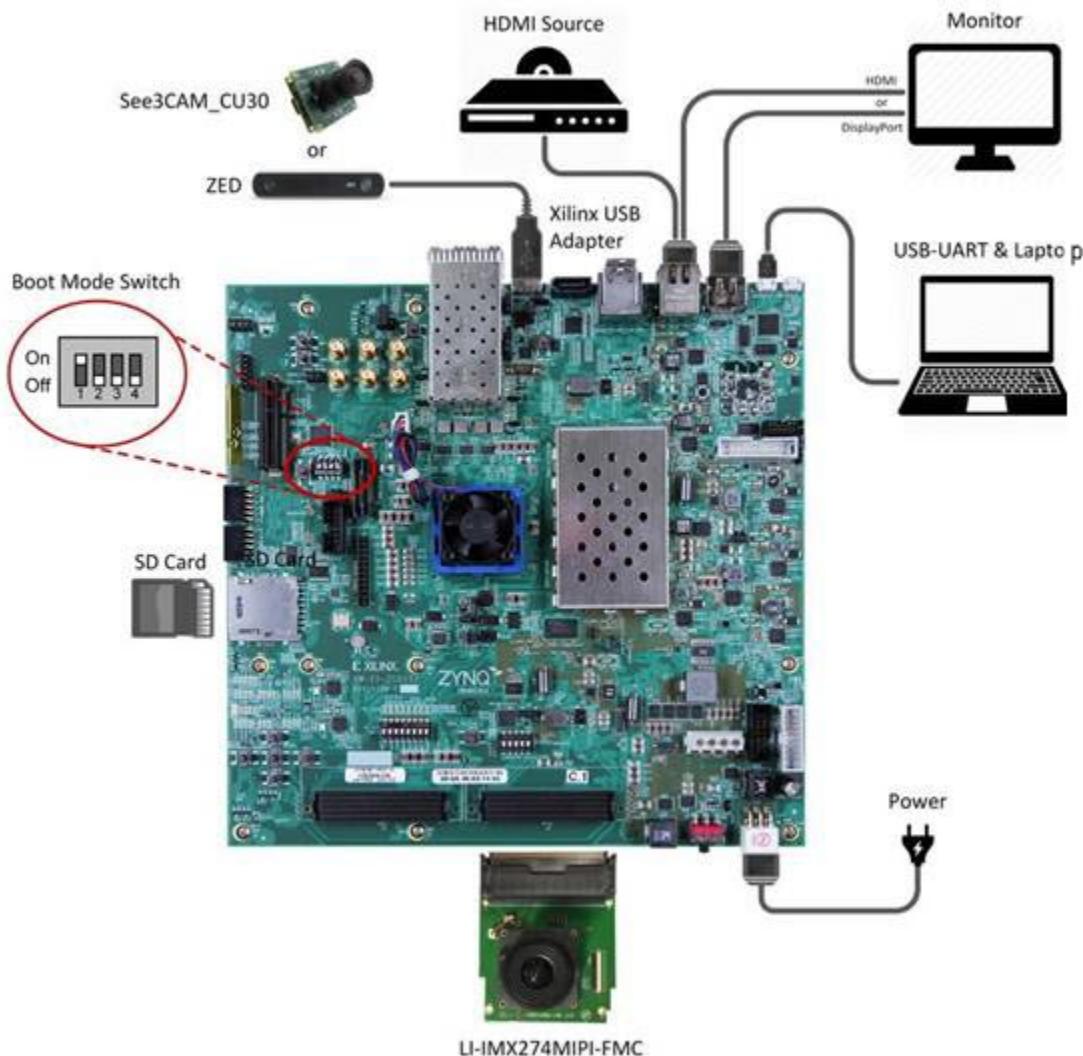
評価ボードの設定

ZCU102/104 評価ボードの設定

ザイリンクスの ZCU102 評価ボードは、ミッドレンジ ZU9 Zynq® UltraScale+™ MPSoC を使用して今すぐ機械学習アプリケーションを開発できるようにサポートします。ZCU102 ボードの詳細は、ザイリンクス ウェブサイトの ZCU102 製品ページ (<https://japan.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html>) を参照してください。

次の図に、ZCU102 の主なコネクティビティ インターフェイスを示します。

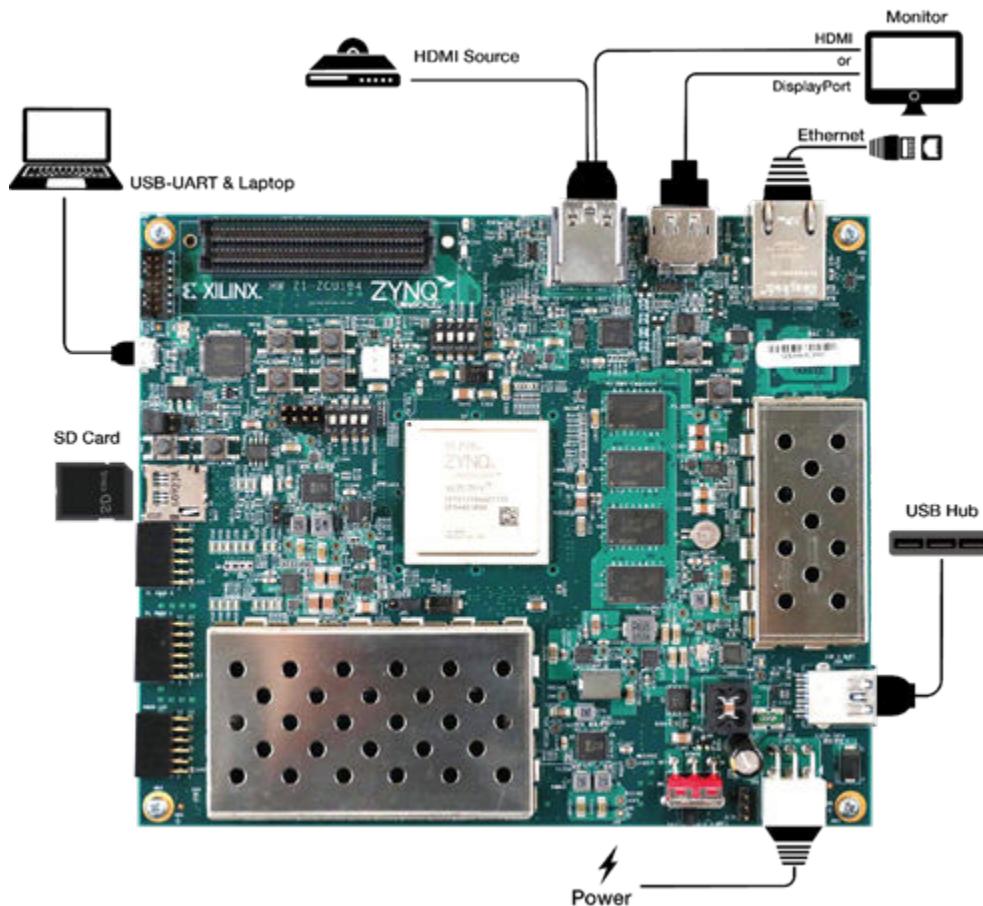
図18: ザイリンクスのZCU102評価ボードとペリフェラルの接続



ザイリンクス ZCU104 評価ボードはミッドレンジの ZU7 Zynq UltraScale+ デバイスを使用しており、機械学習アプリケーションの開発を今すぐ開始できます。ZCU104 ボードの詳細は、ザイリンクス ウェブサイト (<https://japan.xilinx.com/products/boards-and-kits/zcu104.html>) を参照してください。

次の図に、ZCU104 の主なコネクティビティ インターフェイスを示します。

図 19: ザイリンクス ZCU104 評価ボードとペリフェラルの接続

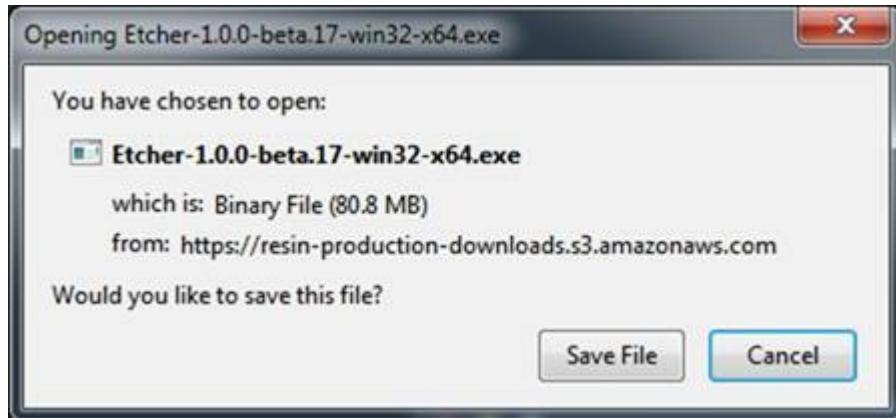


次のセクションでは、評価ボードで ZCU102 を例として使用して、Vitis AI の実行環境をセットアップする手順を説明します。

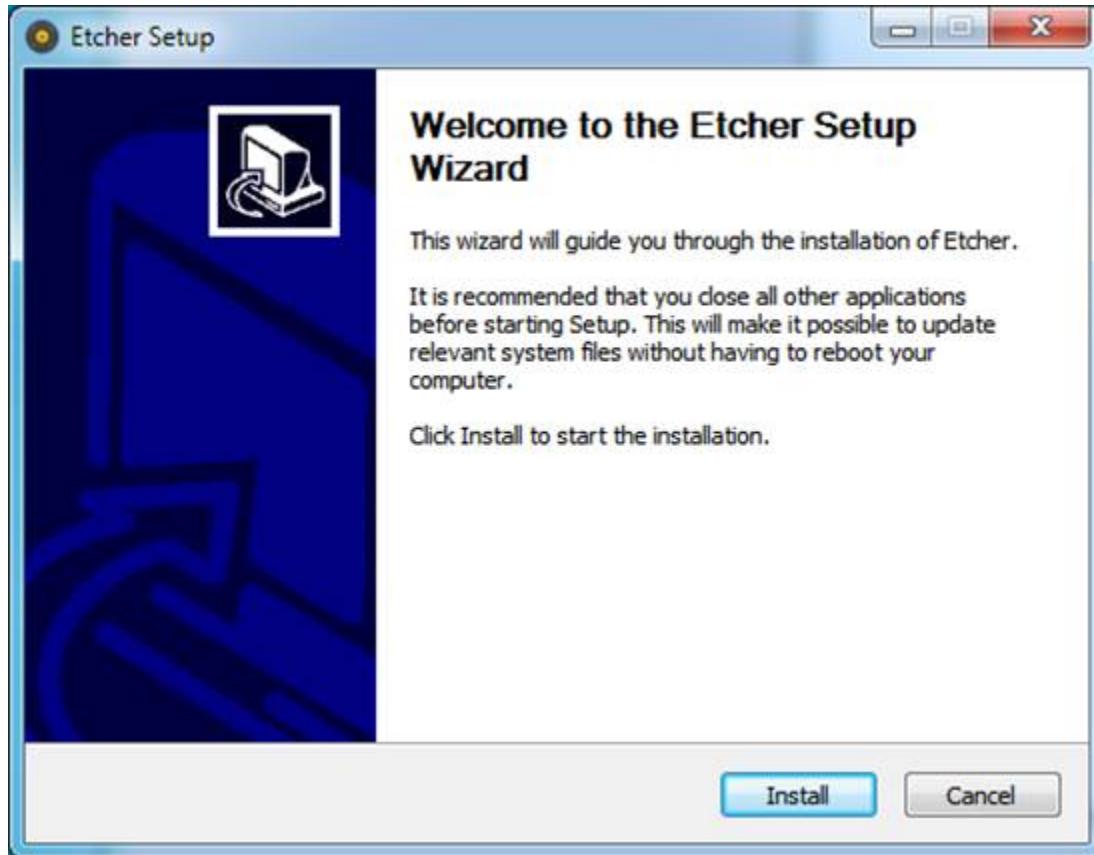
OS イメージを SD カードに書き込む

ZCU102 の場合、システム イメージは [こちら](#) からダウンロードできます。ZCU104 の場合は [こちら](#) からダウンロードできます。SD カードへの書き込みには、Etcher ソフトウェア アプリケーションの使用を推奨します。これは、OS イメージを SD カードに書き込むためのクロスプラットフォーム ツールであり、Windows、Linux、および Mac システムに対応します。ここでは、Windows の場合の例を示します。

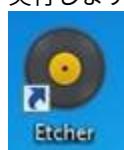
1. <https://etcher.io/> から Etcher をダウンロードして、次の図のようにファイルを保存します。



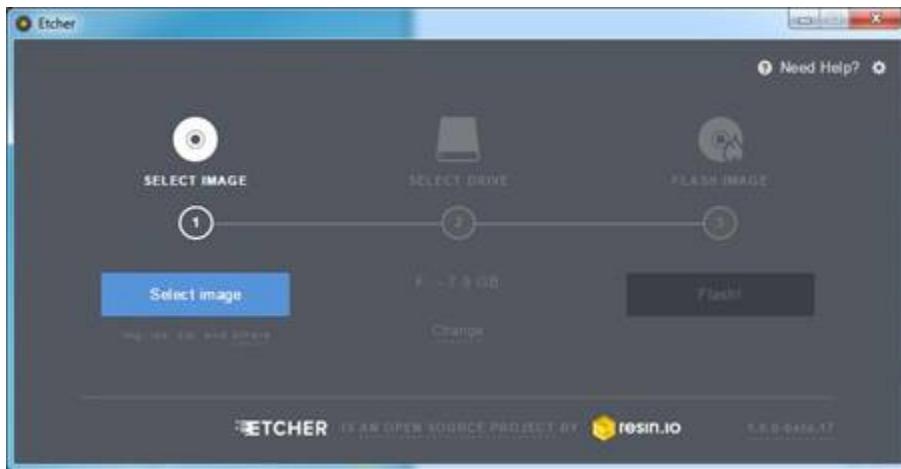
2. 次の図のように Etcher をインストールします。



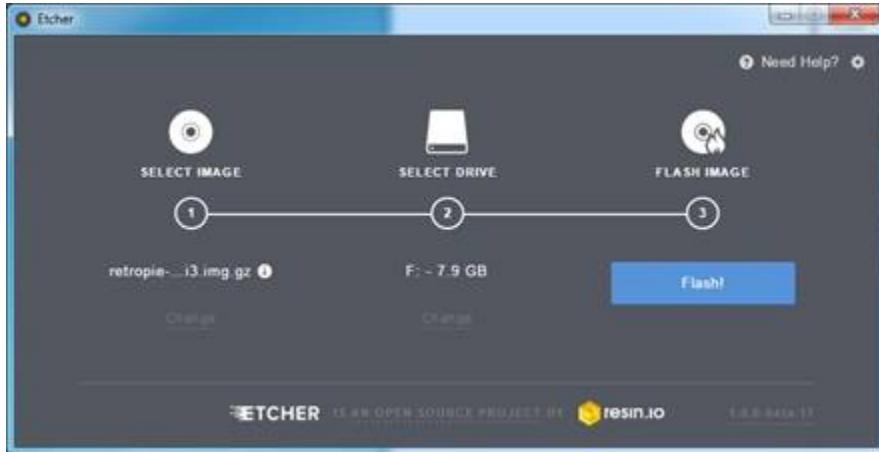
3. USB メモリやバックアップ用ハードディスクなどの外部ストレージ デバイスがある場合は、取り外しておきます。こうすると、SD カードの場所を見つけやすくなります。次に、SD カードをコンピューターのスロットまたは SD カード リーダーに挿入します。
4. 次の図に示す Etcher アイコンをダブルクリック、または [スタート] メニューから選択して Etcher プログラムを実行します。



Etcher が起動します (下図参照)。



5. [Select Image] をクリックして、イメージ ファイルを選択します。[.zip] または [.gz] 形式の圧縮ファイルを選択できます。
6. Etcher が SD ドライブを検出します。ドライブ デスティネーションとイメージ サイズを確認します。
7. [[Flash!]] をクリックします。



評価ボードを起動

Vitis AI ボードを例として使用して、評価ボードを起動する方法を説明します。次の手順に従って、評価ボードを起動します。

1. 電源 (12V ~ 5A) を接続します。
2. UART デバッグインターフェイスをホストへ接続、さらに必要に応じてその他のペリフェラルへ接続します。
3. 電源を入れて、システムを起動します。
4. システムへログインします。
5. システムは、最初の起動時に複数の設定が必要です。設定完了後、これらの設定を有効にするためにボードを再起動します。

評価ボードへのアクセス方法

ZCU102 ボードへアクセスする方法は 3 つあります。

- UART ポート
- イーサネット接続
- スタンドアロン

UART ポート

UART ポートは、ブート プロセスを監視したり、Linux カーネル メッセージを確認するなどデバッグに使用できるほか、ボードにログインする際にも使用できます。ここでは、例として UART のコンフィギュレーション パラメーターを示します。ユーザー名「root」、パスワード「root」でシステムにログインします。

- ボーレート: 115200bps
- データビット: 8
- ストップビット: 1
- パリティなし

注記: Linux システムでは、Minicom を使用してターゲット ボードに直接接続できます。Windows システムでシリアル ポートを介してボードに接続するには、USB-to-UART ドライバーが必要です。

イーサネット インターフェイスを使用

ZCU102 ボードにはイーサネット インターフェイスがあり、SSH サービスがデフォルトで有効になっています。ボードの起動後、SSH クライアントを使用してシステムにログインできます。

`ifconfig` コマンドを使用して UART インターフェイスを介してボードの IP アドレスを設定し、SSH クライアントを使用してシステムにログインします。

ボードをスタンドアロンのエンベデッド システムとして使用

ZCU102 ボードでは、キーボード、マウス、およびモニターを接続できます。正常に起動すると、Linux の GUI デスクトップが表示されます。その後、スタンドアロンのエンベデッド システムとしてボードにアクセスできます。

評価ボードに Vitis AI ランタイムをインストール

ユーザー体験の向上のため、Vitis AI ランタイム パッケージ、VART サンプル、Vitis AI ライブラリ サンプルおよびモデルがボード イメージに組み込まれています。サンプルはコンパイル済みです。したがって、Vitis AI ランタイム パッケージおよびモデル パッケージをボードに別途インストールする必要はありません。ただし、次の手順に従って、モデルまたは Vitis AI ランタイムを独自のイメージまたは公式イメージ上にインストールすることは可能です。

イーサネット接続が確立されたら、GitHub から Vitis™ AI ランタイム (VART) パッケージを評価ボードにコピーし、ZCU102 ボードの Vitis AI 実行環境を設定します。

1. [こちら](#) から `vitis-ai-runtime-1.3.x.tar.gz` をダウンロードします。パッケージを展開し、`scp` を使用して次のファイルをボードにコピーします。

```
tar -xzvf vitis-ai-runtime-1.3.x.tar.gz  
scp -r vitis-ai-runtime-1.3.x/aarch64/centos root@IP_OF_BOARD:~/
```

注記: rpm パッケージは、通常のアーカイブとして取得できます。ライブラリの一部のみ必要な場合は、ホスト側でコンテンツを抽出できます。独立しているのはモデル ライブラリのみであり、その他は共有ライブラリです。操作コマンドは次のとおりです。

```
rpm2cpio libvart-1.3.0-r<x>.aarch64.rpm | cpio -idmv
```

2. SSH を使用してボードにログインします。シリアル ポートを使用してログインすることも可能です。
3. Vitis AI ランタイムをインストールします。次のコマンドを順番に実行します。

```
cd ~/centos
rpm -ivh --force libunilog-1.3.0-r<x>.aarch64.rpm
rpm -ivh --force libxir-1.3.0-r<x>.aarch64.rpm
rpm -ivh --force libtarget-factory-1.3.0-r<x>.aarch64.rpm
rpm -ivh --force libvart-1.3.0-r<x>.aarch64.rpm
```

Vitis AI ライブラリ ベースのサンプルを実行する場合は、次のコマンドを実行して Vitis AI ライブラリ ランタイム パッケージをインストールします。

```
rpm -ivh --force libvitis_ai_library-1.3.0-r<x>.aarch64.rpm
```

インストールが完了すると、`/usr/lib` の下に Vitis AI ランタイム ライブラリがインストールされます。

カスタム ボードの設定

Vitis AI は、公式の ZCU102/ZCU104 ボードとユーザー定義のボードをサポートしています。

カスタム ボードで Vitis AI を実行する場合は、次の手順を順番に実行する必要があります。1 つの手順が完了したことを確認してから、次の手順に進んでください。

1. カスタム ボードのプラットフォーム システムを作成します。詳細は、『Vitis 統合ソフトウェア プラットフォーム の資料: エンベデッド ソフトウェア開発』(UG1400: [英語版](#)、[日本語版](#)) および https://github.com/Xilinx/Vitis_Embedded_Platform_Source/tree/master/Xilinx_Official_Platforms を参照してください。
2. DPU IP を統合します。DPU IP を統合する方法については、<https://github.com/Xilinx/Vitis-AI/tree/master/dsa/DPU-TRD> を参照してください。

注記: この手順が完了すると、`sd_card` ディレクトリと、DPU を含む `sd_card.img` イメージが作成されます。既知の問題については、[既知の問題](#) を参照してください。

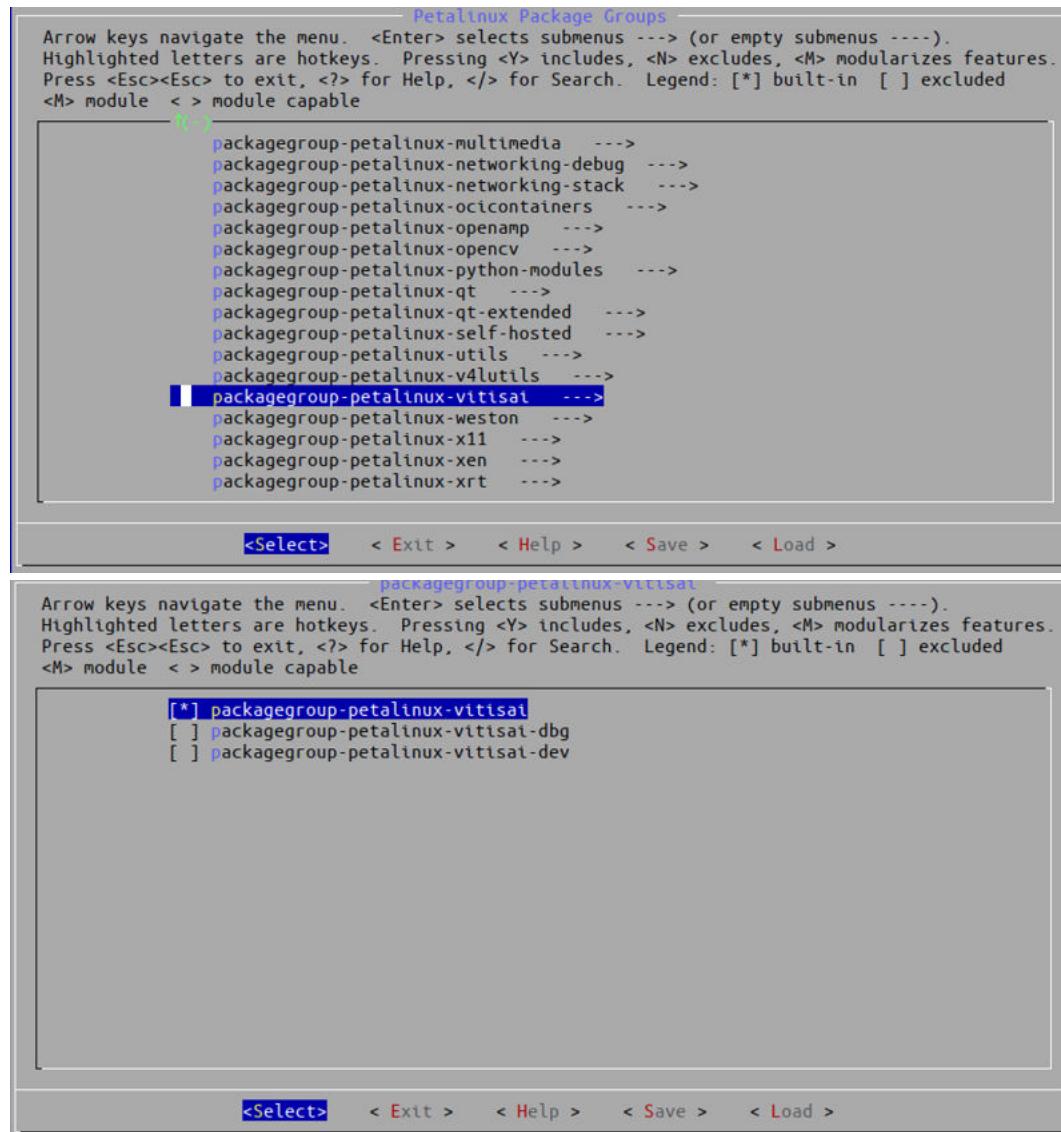
3. Vitis AI の依存ライブラリをインストールします。

Vitis AI の依存ライブラリをインストールするには 2 つの方法があります。1 つは PetaLinux のコンフィギュレーションによってシステムを再構築する方法で、もう 1 つは Vitis AI 依存ライブラリをオンラインでインストールする方法です。

- a. PetaLinux のコンフィギュレーションによってシステムを再構築するには、次のコマンドを実行します。

```
petalinux-config -c rootfs
```

- b. 次の図に示すように、[packagegroup-petalinux-vitisai] が選択されていることを確認します。



- c. 次のコマンドを実行して、システムを再コンパイルします。

```
petalinux-build
```

- d. 次のコードに示すように、`dnf install packagegroup-petalinux-vitisai` を実行してインストールを完了することにより、Vitis AI 依存ライブラリをオンラインでインストールすることもできます。

```
root@xilinx-zcu104-2020_1:/media/sd-mmcblk0p1# dnf install
packagegroup-petalinux-vitisai
Last metadata expiration check: 1 day, 18:12:25 ago on Wed Jun 17
09:35:01 2020.
Package packagegroup-petalinux-vitisai-1.0-r0.noarch is already
installed.
Dependencies resolved.
Nothing to do.
Complete!
```

注記: この方法を使用する場合は、ボードがインターネットに接続されていることを確認します。

注記: この手順が完了すると、VART の 1.2 バージョンもシステムにインストールされ、Vitis AI 用のシステム イメージが使用可能になります。

注記: ターゲット上でこのサンプルをコンパイルする場合は、`packagegroup-petalinux-vitisai-dev` および `packagegroup-petalinux-self-hosted` を選択して、システムを再コンパイルします。

4. イメージを SD カードに書き込みます。

SD カードへ新しいイメージを書き込む方法については、[OS イメージを SD カードに書き込む](#) を参照してください。

5. 必要に応じて、Vitis AI ランタイム ライブラリを VAI1.3 にアップデートします。

上記のシステム イメージでカスタム ボードを起動すると、VART の 1.2 バージョンがインストールされます。1.3 バージョンにアップデートする場合は、次に示す 5 つのライブラリ パッケージをアップデートする必要があります。

- libunilog
- libxir
- libtarget-factory
- libvart
- libvitis_ai_library

Vitis AI ランタイム ライブラリのインストール方法については、[評価ボードに Vitis AI ランタイムをインストール](#) を参照してください。

Vitis AI ランタイムをインストールすると、次に示すように、`dpu.xclbin` ファイルの位置を示す `vart.conf` ファイルが `/etc` の下に生成されます。Vitis AI サンプルは、`vart.conf` ファイルを読み出すことにより、`dpu.xclbin` ファイルをフェッチします。ボード上の `dpu.xclbin` ファイルの位置がデフォルトと異なる場合は、`vart.conf` 内の `dpu.xclbin` のパスを変更します。

```
root@xilinx-zcu102-2020_1:/etc# cat /etc/vart.conf
firmware: /media/sd-mmcblk0p1/dpu.xclbin
```

注記: この手順により、Vitis AI サンプルを実行できるシステムが生成されます。

6. Vitis AI サンプルを実行します。Vitis AI サンプルの実行方法については、[サンプルを実行](#) を参照してください。

サンプルを実行

Vitis AI 開発キット v1.3 リリースでは、2 種類のサンプルが提供されています。サンプルは次のとおりです。

- VART ベースのサンプルでは、クラウドからエッジまで対応できる Vitis AI の統合された高レベル C++/Python API の使用方法を示しています。
- DNNDK ベースのサンプルでは、エッジ DPUCZDX8G でのみ利用可能な Vitis AI の高度な低レベル C++/Python API の使用方法を示しています。

これらのサンプルは <https://github.com/xilinx/vitis-ai> から入手できます。ザイリンクス ZCU102 および ZCU104 ボードを使用してサンプルを実行する場合、すべてのサンプルは X11 が適切に機能していることが条件であるため、SSH ターミナルを使用してボードにログインする際に「`ssh -X`」オプションまたは `export DISPLAY=192.168.0.10:0.0` (ホスト マシンの IP アドレスが 192.168.0.10 の場合) コマンドを使用して X11 転送を有効にしてください。

注記: X11 がサポートされていない場合、サンプルは UART ポート経由で機能しません。オプションとして、イーサネットを使用する代わりに、ボードをモニターに直接接続できます。

Vitis AI のサンプル

Vitis AI は、C++ および Python のサンプル デザインを提供し、統合されたクラウド/エッジ ランタイム プログラミング API の使用法を示します。

注記: サンプル コードは、ユーザーが新しいランタイム (VART) を使用できるようにサポートすることを目的としており、パフォーマンスのベンチマーク用ではありません。

統合された API に慣れるために、VART のサンプルを利用できます。これらのサンプルは、API を理解するためのものであり、高性能を実証するためのものではありません。これらの API はエッジとクラウド間で互換性がありますが、クラウド ボードにはバッチ処理などのさまざまなソフトウェア最適化があり、エッジではより高性能を実現するためにマルチスレッドが必要です。より高い性能を求める場合は、Vitis AI ライブラリのサンプルおよびデモ ソフトウェアへアクセスしてください。

より高性能を追求するために最適化を実行する場合は、次のような推奨事項があります。

- スレッドのパイプライン構造を再配置して、すべての DPU スレッドに独自の「DPU」ランナー オブジェクトが含まれるようにします。
- DPU FPS が表示レートよりも高い場合には、表示スレッドを最適化して一部のフレームをスキップします。ビデオ表示に 200FPS は高過ぎます。
- 事前のデコード。ビデオ ファイルは H.264 でエンコードされている場合が多く、デコーダーは DPU よりも低速で、CPU リソースを大量に消費します。ビデオ ファイルは、あらかじめデコードして未加工の形式に変換しておく必要があります。
- Alveo ボード上のバッチ モードでは、ビデオ フレームにジッターが発生する可能性があるため、特に配慮する必要があります。ZCU102 ではバッチ モードをサポートしていません。
- OpenCV cv::imshow は低速であるため、libdrm.so を使用する必要があります。これは、X サーバー経由ではなく、ローカル ディスプレイ専用です。

次の表では、これらの Vitis AI のサンプルについて説明しています。

表 3: Vitis AI のサンプル

ID	サンプル名	モデル	フレームワーク	注記
1	resnet50	ResNet-50	Caffe	Vitis AI の統合された C++ API を使用する画像分類
2	resnet50_mt_py	ResNet-50	TensorFlow	Vitis AI の統合された Python API を使用するマルチスレッドの画像分類
3	inception_v1_mt_py	Inception-v1	TensorFlow	Vitis AI の統合された Python API を使用するマルチスレッドの画像分類
4	pose_detection	SSD、姿勢検出	Caffe	Vitis AI の統合された C++ API を使用する姿勢検出
5	video_analysis	SSD	Caffe	Vitis AI の統合された C++ API を使用するトラフィック検知
6	adas_detection	YOLOv3	Caffe	Vitis AI の統合された C++ API を使用する ADAS 検知
7	segmentation	FPN	Caffe	Vitis AI の統合された C++ API を使用するセマンティック セグメンテーション
8	squeezeenet_pytorch	SqueezeNet	PyTorch	Vitis AI の統合された C++ API を使用する画像分類

Vitis AI の統合された C++ 高レベル API を使用してモデルを運用する際の一般的なコード スニペットを次に示します。

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

Vitis AI の統合された Python 高レベル API を使用してモデルを運用する際の一般的なコード スニペットを次に示します。

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

DPUCZD8G および DPUCAHX8H で Vitis AI サンプルを実行

エッジまたはクラウドで Vitis™ AI サンプルを実行する前に、[こちら](#) から

vitis_ai_runtime_r1.3.0_image_video.tar.gz をダウンロードしてください。次のサンプルで使用する画像とビデオは、パッケージに含まれます。

ユーザー体験の向上のため、Vitis AI ランタイム パッケージ、VART サンプル、Vitis AI ライブラリ サンプルおよびモデルはボード イメージに組み込まれ、サンプルはコンパイル済みとなっています。サンプル プログラムはターゲット上で直接実行できます。

エッジの場合 (DPUCZDX8G)

1. 次のコマンドで scp を使用して、サンプルとホストからターゲットへ

vitis_ai_runtime_r1.3.0_image_video.tar.gz をダウンロードします。

```
scp vitis_ai_runtime_r1.3.0_image_video.tar.gz root@[IP_OF_BOARD]:~/
```

2. vitis_ai_runtime_r1.3.0_image_video.tar.gz パッケージを解凍します。

```
tar -xzvf vitis_ai_runtime_r1.3.0_image_video.tar.gz -C ~/Vitis-AI/demo/
VART
```

3. モデルをダウンロードします。モデルのダウンロード リンクは、モデルの yaml ファイルに記述されています。Vitis-AI/models/AI-Model-Zoo で yaml ファイルを見つけて、対応するプラットフォームのモデルをダウンロードできます。ここでは、resnet50 の例を示します。

```
wget https://www.xilinx.com/bin/public/openDownload?filename=resnet50-
zcu102_zcu104-r1.3.0.tar.gz -O resnet50-zcu102_zcu104-r1.3.0.tar.gz
```

```
scp resnet50-zcu102_zcu104-r1.3.0.tar.gz root@[IP_OF_BOARD]:~/
```

4. ターゲット上でモデルを解凍し、インストールします。

注記: /usr/share/vitis_ai_library/models フォルダーが存在しない場合は、先にこのフォルダーを作成します。

```
mkdir -p /usr/share/vitis_ai_library/models
```

次のコマンドを実行して、モデル パッケージをインストールします。

```
tar -xzvf resnet50-zcu102_zcu104-r1.3.0.tar.gz
cp resnet50 /usr/share/vitis_ai_library/models -r
```

- ターゲット ボードでサンプルのディレクトリに移動します。ここでは、resnet50 の例を示します。

```
cd ~/Vitis-AI/demo/VART/resnet50
```

- サンプル デザインを実行します。

```
./resnet50 /usr/share/vitis_ai_library/models/resnet50/resnet50.xmodel
```

注記: 上記の実行可能なプログラムが存在しない場合は、ホスト上でプログラムをクロス コンパイルします。

注記: ビデオ入力のサンプルの場合、公式システム イメージには「webm」および「raw」形式のみがデフォルトでサポートされます。その他の形式のビデオ データを使用する場合は、関連するパッケージをシステムにインストールします。

次の表に、すべての Vitis AI サンプルの実行コマンドを示します。

表 4: ZCU102/ZCU104 における Vitis AI サンプルの起動コマンド

ID	サンプル名	コマンド
1	resnet50	./resnet50 /usr/share/vitis_ai_library/models/resnet50/resnet50.xmodel
2	resnet50_mt_py	python3 resnet50.py 1 /usr/share/vitis_ai_library/models/resnet50/resnet50.xmodel
3	inception_v1_mt_py	python3 inception_v1.py 1 /usr/share/vitis_ai_library/models/inception_v1_tf/inception_v1_tf.xmodel
4	pose_detection	./pose_detection video/pose.webm /usr/share/vitis_ai_library/models/sp_net/sp_net.xmodel /usr/share/vitis_ai_library/models/ssd_pedestrian_pruned_0_97/ssd_pedestrian_pruned_0_97.xmodel
5	video_analysis	./video_analysis video/structure.webm /usr/share/vitis_ai_library/models/ssd_traffic_pruned_0_9/ssd_traffic_pruned_0_9.xmodel
6	adas_detection	./adas_detection video/adas.webm /usr/share/vitis_ai_library/models/yolov3_adas_pruned_0_9/yolov3_adas_pruned_0_9.xmodel
7	segmentation	./segmentation video/traffic.webm /usr/share/vitis_ai_library/models/fpn/fpn.xmodel
8	squeezezenet_pytorch	./squeezezenet_pytorch /usr/share/vitis_ai_library/models/squeezezenet_pt/squeezezenet_pt.xmodel

クラウドの場合 (DPUCAHX8H)

クラウド上でサンプルを実行する前に、U50、U50LV、または U280 などの Alveo カードがサーバーにインストールされており、Docker システムが読み込まれて動作していることを確認してください。

Vitis-AI のダウンロード後、Vitis-AI ディレクトリに移動し、Docker を起動します。

VART のサンプルは Docker システム内の /workspace/demo/VART/ のパスにあります。

- vitis_ai_runtime_r1.3.0_image_video.tar.gz パッケージをダウンロードして、解凍します。

```
tar -xzvf vitis_ai_runtime_r1.3.0_image_video.tar.gz -C /workspace/demo/VART
```

2. サンプルをコンパイルします(resnet50 を例として使用)。

```
cd /workspace/demo/VART/resnet50
bash -x build.sh
```

コンパイルが完了すると、実行可能ファイル resnet50 が現在のディレクトリに生成されます。

3. モデルをダウンロードします。モデルのダウンロード リンクは、モデルの yaml ファイルに記述されています。Vitis-AI/models/AI-Model-Zoo で yaml ファイルを見つけて、対応するプラットフォームのモデルをダウンロードできます。ここでは、resnet50 の例を示します。

```
wget https://www.xilinx.com/bin/public/openDownload?filename=resnet50-u50-r1.3.0.tar.gz -O resnet50-u50-r1.3.0.tar.gz
```

4. ターゲット上でモデルを解凍し、インストールします。

/usr/share/vitis_ai_library/models フォルダーが存在しない場合は、先にこのフォルダーを作成します。

```
sudo mkdir -p /usr/share/vitis_ai_library/models
```

モデル パッケージをインストールします。

```
tar -xzvf resnet50-u50-r1.3.0.tar.gz
sudo cp resnet50 /usr/share/vitis_ai_library/models -r
```

5. サンプルを実行します。

```
./resnet50 /usr/share/vitis_ai_library/models/resnet50/resnet50.xmodel
```

次の表に、クラウド内のすべての Vitis AI サンプルの実行コマンドを示します。

表 5: U50/U50LV/U280 における Vitis AI サンプルの起動コマンド

ID	サンプル名	コマンド
1	resnet50	./resnet50 /usr/share/vitis_ai_library/models/resnet50/resnet50.xmodel
2	resnet50_mt_py	/usr/bin/python3 resnet50.py 1 /usr/share/vitis_ai_library/models/resnet50/resnet50.xmodel
3	inception_v1_mt_py	/usr/bin/python3 inception_v1.py 1 /usr/share/vitis_ai_library/models/inception_v1_tf/inception_v1_tf.xmodel
4	pose_detection	./pose_detection video/pose.mp4 /usr/share/vitis_ai_library/models/sp_net/sp_net.xmodel /usr/share/vitis_ai_library/models/ssd_pedestrian_pruned_0_97/ssd_pedestrian_pruned_0_97.xmodel
5	video_analysis	./video_analysis video/structure.mp4 /usr/share/vitis_ai_library/models/ssd_traffic_pruned_0_9/ssd_traffic_pruned_0_9.xmodel
6	adas_detection	./adas_detection video/adas.avi /usr/share/vitis_ai_library/models/yolov3_adas_pruned_0_9/yolov3_adas_pruned_0_9.xmodel
7	segmentation	./segmentation video/traffic.mp4 /usr/share/vitis_ai_library/models/fpn/fpn.xmodel
8	squeezezenet_pytorch	./squeezezenet_pytorch /usr/share/vitis_ai_library/models/squeezezenet_pt/squeezezenet_pt.xmodel

サポート

ディスカッション、知識の共有、FAQ、およびテクニカル サポートのリクエストについては、ザイリンクス ウェブサイトの [Vitis AI ライブラリ コミュニティ フォーラム](#) にアクセスしてください。

Vitis AI Model Zoo ネットワークの理解

Vitis™ AI Model Zoo には、ザイリンクス プラットフォームで深層学習推論アプリケーションをすばやく運用するための、最適化済み深層学習モデルが含まれています。これらのモデルは、ADAS/AD、ビデオ監視、ロボット工学、データセンターなどのさまざまなアプリケーション分野に対応します。これらの無償の学習済みモデルを利用することで、深層学習の高速化が可能になります。

Vitis 1.3 AI Model Zoo では、各種のニューラル ネットワーク モデルが、広く使用されている 3 つのフレームワーク (Caffe、TensorFlow、PyTorch) と共に提供されます。各モデルについて、モデル名、フレームワーク、タスクのタイプ、ネットワーク バックボーン、トレーニングおよび検証用データセット、浮動小数点 OP、ブルーニングの有無、ダウンロード リンク、ライセンス、および md5 チェックサムを記述する .yaml ファイルが公開されています。Vitis 1.3 AI Model Zoo 内のモデル リストを参照して、関心のあるニューラル ネットワーク モデルを選択し、指定した .yaml ファイルからモデルの基本情報を取得できます。モデルは、.yaml ファイル内のダウンロード リンクから無償でダウンロードできます。

たとえば、TensorFlow での汎用画像分類に使用される ResNet-50 モデルが必要な場合は、`tf_resnetv1_50_imagenet_224_224_6.97G_1.3` という名前のモデルを探します。標準命名規則に従って、各モデルの名前は `F_M_(D)_H_W_(P)_C_V` の形式をとります。

- F はトレーニング フレームワークを指定します。cf は caffe、tf は TensorFlow、dk は Darknet、pt は PyTorch です。
- M はモデルの機能を指定します。
- D はデータセットを指定します。これはデータセットがパブリックかプライベートかによって省略可能です。Mixed は、複数のパブリック データセットの混合を意味します。
- H は入力データの高さを指定します。
- W は入力データの幅を指定します。
- P はブルーニング率を指定します。これは計算がどれだけ削減されるかを意味します。ブルーニングしない場合は省略可能です。
- C はモデルの計算量 (画像 1 個あたりの Gops 値) を指定します。
- V は Vitis AI のバージョンを指定します。

つまり、`tf_resnetv1_50_imagenet_224_224_6.97G_1.3` は、Imagenet データセットを使用して TensorFlow でトレーニングされた ResNet v1-50 モデルであり、入力データ サイズは 224*224、ブルーニングなし、画像 1 個あたりの計算量は 6.97Gops、Vitis AI のバージョンは 1.3 です。

このモデルを選択し、`tf_resnetv1_50_imagenet_224_224_6.97G_1.3.yaml` に含まれるリンクから手動で、または .yaml の情報を読み出せるツールを使用して、モデルをダウンロードできます。

モデル リストの詳細は、<https://github.com/Xilinx/Vitis-AI/tree/master/models/AI-Model-Zoo/model-list> を参照してください。

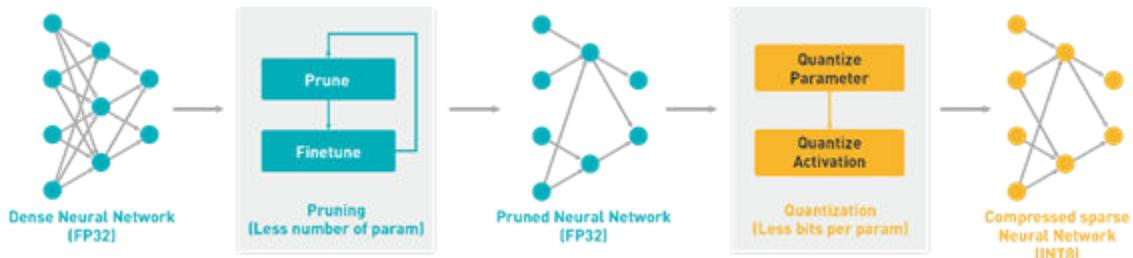
モデルの量子化

概要

推論プロセスでは膨大な演算が発生するため、エッジ アプリケーションの低レイテンシかつ高スループットという要件を満たすには、広いメモリ帯域幅が必要です。

これらの要件には、量子化とチャネル プルーニング技術を使用することで、最小限の精度の低下で高い性能と電力効率を実現しながら対応できます。量子化によって、整数計算ユニットの利用や、下位ビットで重みやアクティベーションを表すことが可能になり、プルーニングで全体的な計算量を削減します。Vitis™ AI クオンタイザーには、量子化ツールのみが含まれています。プルーニング ツールは、Vitis AI オプティマイザーにパッケージ化されています。プルーニング ツールが必要な場合は、Vitis AI 開発キットのサポートチームへお問い合わせください。

図 20: プルーニングと量子化フロー



一般に、ニューラル ネットワークの学習では 32 ビット浮動小数点の重みと活性化値を使用します。Vitis AI クオンタイザーは、32 ビット浮動小数点の重みやアクティベーション コードを 8 ビットの整数 (INT8) に変換することで、予測精度を損なうことなく計算の複雑レベルを軽減できます。固定小数点ネットワーク モデルの方が、浮動小数点モデルよりも必要なメモリ帯域幅が狭く、速度と電力効率が向上します。Vitis AI クオンタイザーは、たたみ込み、プーリング、完全接続、バッチ正規化など、ニューラル ネットワークの一般的なレイヤーをサポートします。

Vitis AI クオンタイザーは、現在 TensorFlow (1.x および 2.x の両方)、PyTorch、および Caffe をサポートしています。クオンタイザーの名前は、それぞれ vai_q_tensorflow、vai_q_pytorch、および vai_q_caffe です。TensorFlow 1.x と TensorFlow 2.x 用の Vitis AI クオンタイザーは、異なる方法で実装され、個別にリリースされています。TensorFlow 1.x 用の Vitis AI クオンタイザーは、TensorFlow 1.15 に基づいています。量子化機能の追加後、Vitis AI クオンタイザーはスタンドアロン パッケージを再構築および再分配します。TensorFlow 2.x 用の Vitis AI クオンタイザーは、複数の量子化 API を備えた Python パッケージです。このパッケージをインポートすると、Vitis AI クオンタイザーを TensorFlow のプラグインのように動作させることができます。

表 6: Vitis AI クオンタイザーでサポートされるフレームワークおよび機能

	バージョン	機能		
		量子化キャリブレーション (トレーニング後の量子化)	量子化微調整 (QAT (量子化認識トレーニング))	高速微調整 (高度なキャリブレーション)
TensorFlow 1.x	1.15 に基づく	あり	あり	なし
TensorFlow 2.x	2.3 をサポート	あり	あり	なし。この機能は現在開発中です。
PyTorch	1.2 ~ 1.4 をサポート	あり	あり	あり
Caffe	-	あり	あり	なし

量子化キャリブレーションプロセスでは、アクティベーション分布を分析するために必要なのはラベル付けされていない小規模な画像セットのみです。量子化キャリブレーションの実行時間は、ニューラルネットワークのサイズによって数秒から数分とさまざまです。一般的に、量子化後には精度がわずかに低下します。ただし、Mobilenetなどの一部のネットワークでは、精度の損失が大きくなる場合があります。このような場合、微調整を実行して量子化モデルの精度をさらに向上させることができます。量子化の微調整には、オリジナルのトレーニングデータセットが必要です。試験によると、数エポックの微調整が必要で、微調整時間は数分から数時間まで変化します。量子化微調整を実行する際は、学習レートを低く抑えることを推奨します。

量子化キャリブレーション用に、Vitis AI 1.3 にはクロスレイヤーイコライゼーション¹アルゴリズムが実装されています。クロスレイヤーイコライゼーションにより、特に depthwise たたみ込みを含むネットワークで、キャリブレーション性能が向上します。

AdaQuant アルゴリズム²は、ラベルなしのデータの小さな集合を使用して、アクティベーションを調整するだけでなく、重みを微調整します。AdaQuant は、キャリブレーションと同じようにラベルなしデータの小さな集合を使用しますが、微調整のようにモデルを変更します。Vitis AI クオンタイザーはこのアルゴリズムを実装しており、「高速微調整」または「高度なキャリブレーション」と呼んでいます。高速微調整では、量子化キャリブレーションより精度が向上しますが、多少低速になります。高速微調整では、各実行で結果が多少異なりますが、この違いは無視してかまいません。これは量子化微調整の場合と同じです。

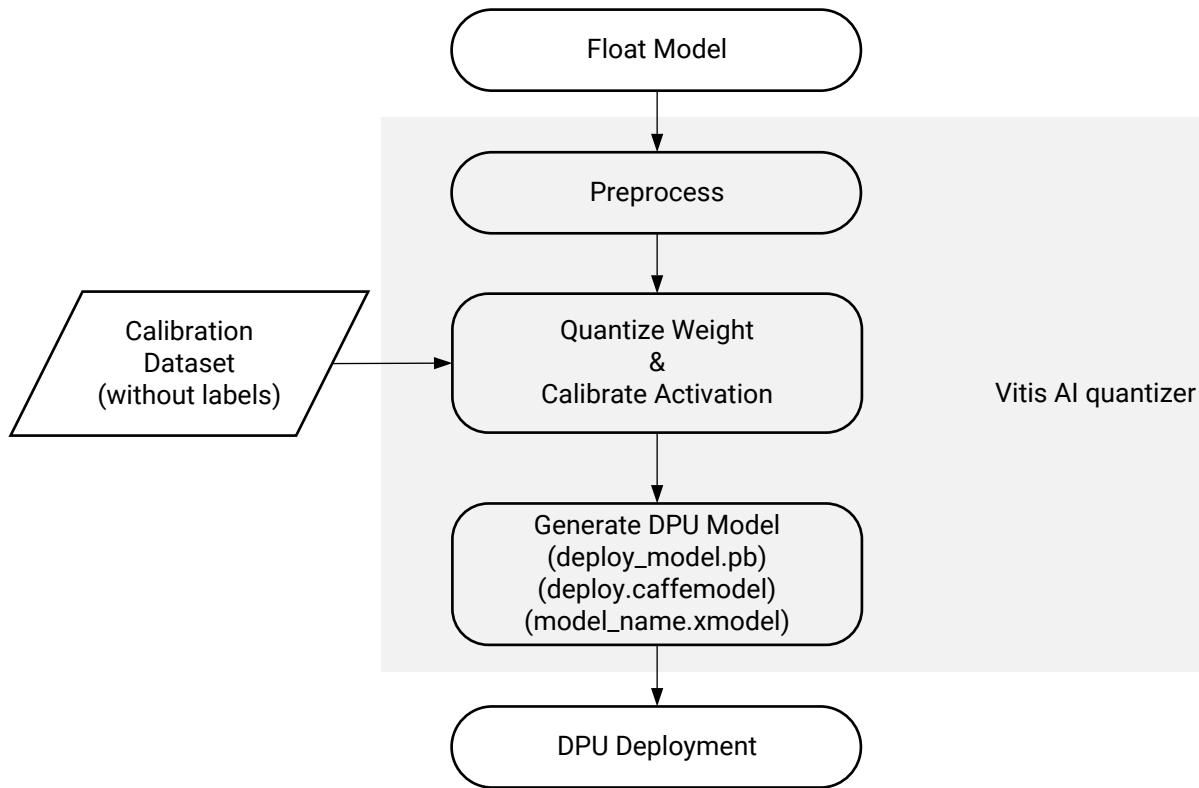
注記:

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

Vitis AI クオンタイザーのフロー

モデル全体の量子化フローを次の図に示します。

図21: VAI クオンタイザーのワークフロー



X24603-121020

Vitis AI クオンタイザーは、入力として浮動小数点モデルを取得し、前処理(バッチ正規化の折りたたみと不要なノードの削除)を実行して、指定されたビット幅に重み/バイアスとアクティベーションを量子化します。

アクティベーションの統計データを取得し、量子化されたモデルの精度を向上させるために、Vitis AI クオンタイザーは、推論の反復を複数回実行し、アクティベーションを調整する必要があります。したがって、キャリブレーションイメージデータセットの入力が必要になります。一般的に 100 ~ 1000 個のキャリブレーションイメージでクォンタイザーは機能します。この場合、バックプロパゲーション(誤差逆伝搬法)の必要がないため、ラベルのないデータセットで十分です。

キャリブレーション後、量子化されたモデルは、DPU のデータ形式に従って DPU で運用可能なモデル(vai_q_tensorflow の場合は deploy_model.pb、vai_q_pytorch の場合は model_name.xmodel、vai_q_caffe の場合は deploy.prototxt/deploy.caffemodel)に変換されます。このモデルは、その後 Vitis AI コンパイラによってコンパイルされ、DPU にデプロイされます。量子化されたモデルは、TensorFlow、PyTorch、または Caffe フレームワークの標準バージョンで取り込むことはできません。

TensorFlow 1.x バージョン (vai_q_tensorflow)

vai_q_tensorflow のインストール

vai_q_tensorflow をインストールする方法は 2 つあります。

Docker コンテナーを使用してインストール

Vitis AI は、vai_q_tensorflow を含む量子化ツールの Docker コンテナーを提供します。コンテナーの実行後、Conda 環境「vitis-ai-tensorflow」をアクティベートします。

```
conda activate vitis-ai-tensorflow
```

ソース コードを使用してインストール

vai_q_tensorflow は、TensorFlow のブランチ「r1.15」からのフォークです。vai_q_tensorflow は Vitis AI クオンタイザ内 のオープンソースです。vai_q_tensorflow の構築プロセスは、TensorFlow 1.15 と同じです。詳細は、『TensorFlow の資料』を参照してください。

vai_q_tensorflow の実行

浮動小数点モデルと関連する入力ファイルを準備する

vai_q_tensorflow を実行する前に、次の表にリストされているファイルを含む、浮動小数点形式の凍結された推論 TensorFlow モデルとキャリブレーション セットを準備します。

表 7: vai_q_tensorflow 用の入力ファイル

No.	名前	説明
1	frozen_graph.pb	浮動小数点の凍結された推論グラフ。トレーニング グラフではなく推論グラフであることを確認する必要があります。
2	キャリブレーション データセット	100 ~ 1000 個のイメージを含むトレーニング データセットの一部。
3	input_fn	量子化の実行中にキャリブレーション データセットを frozen_graph の入力データに変換する入力関数。通常はデータの前処理と拡張を実行します。

凍結された推論グラフを取得

ほとんどの場合、TensorFlow 1.x を使用してモデルをトレーニングすると、GraphDef ファイル(通常は .pb または .pbtxt 拡張子で終わる)とチェックポイント ファイル セットを格納するフォルダーが作成されます。モバイルまたは組み込みシステムでの運用に必要なのは、「凍結」された単一の GraphDef ファイルです。すべての変数をインライン定数に変換しているため、すべて 1 つのファイルに含めることができます。変換処理用として、TensorFlow は freeze_graph.py を提供します。これは vai_q_tensorflow クオンタイザ ツールと共に自動的にインストールされます。

コマンド ラインの使用例は次のとおりです。

```
$ 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 は、トレーニング グラフ以外の推論グラフである必要があります。データ プリプロセッシング処理と損失関数は推論と運用には不要なため、frozen_graph.pb にはモデルの主要な部分のみが含まれます。特にデータ プリプロセッシング処理は、量子化の実行中に Input_fn によって行われ、キャリブレーション用に適切な入力データが生成されます。

注記: ドロップアウトやバッチ正規化などの一部の動作は、トレーニングフェーズと推論フェーズで異なる動作をします。グラフを凍結するときは、それらが推論フェーズであることを確認してください。たとえば、`tf.layers.dropout`/`tf.layers.batch_normalization`を使用する際にフラグ `is_training=False` を設定できます。`tf.keras`を使用するモデルの場合、グラフを作成する前に `tf.keras.backend.set_learning_phase(0)` を呼び出します。



ヒント: オプションの詳細は、`freeze_graph --help` と入力します。

入力ノード名と出力ノード名はモデルによって異なりますが、`vai_q_tensorflow` クオンタイザーを使用して調査および推定できます。例として、次のコード スニペットを参照してください。

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

グラフ内にグラフ用の前処理や後処理がある場合、推定された入力ノードと出力ノードは量子化に使用できません。これは、一部の動作が量子化できないため、量子化したモデルを DPU で運用する場合に、Vitis AI コンパイラでコンパイルするとエラーが発生する可能性があるためです。

グラフの入力および出力名を取得するもう 1 つの方法は、グラフを可視化することです。これには、TensorBoard と Netron を使用できます。次の例は Netron を使用しています。

```
$ pip install netron
$ netron /tmp/inception_v3_inf_graph.pb
```

キャリブレーションデータセットと入力関数を取得

通常は、学習/検証用のデータセットまたは実際のアプリケーションで用いる画像のサブセット（性能を維持するために少なくとも 100 枚以上）をキャリブレーションセットとして使用します。入力関数とは、キャリブレーションデータセットをロードしてデータの前処理を実行するために Python でインポート可能な関数です。`vai_q_tensorflow` クオンタイザーは、グラフに保存されていない前処理を実行するために `input_fn` を受け入れることができます。前処理のサブグラフが凍結されたグラフに保存されている場合、`input_fn` はデータセットからイメージを読み取り、`feed_dict` を返すだけです。

入力関数の形式は、`module_name.input_fn_name` です（例：`my_input_fn.calib_input`）。`input_fn` は、キャリブレーションのステップ番号を示す `int` オブジェクトを入力として受け取り、各呼び出しに対して `dict("placeholder_name", numpy.Array")` オブジェクトを返します。これは推論実行時にモデルのプレースホルダーノードに提供されます。`numpy.array` の形状は、プレースホルダーと一致している必要があります。次の擬似コードの例を参照してください。

```
$ "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}
```

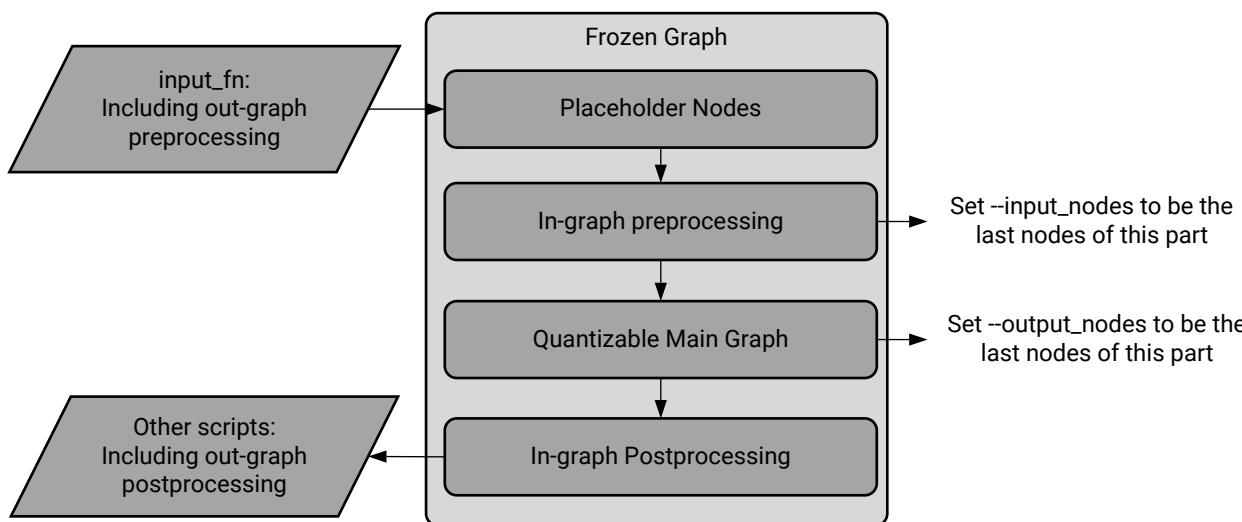
vai_q_tensorflow を使用したモデルの量子化

次のコマンドを実行してモデルを量子化します。

```
$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` と `output_nodes` の引数は、量子化グラフの入力ノード名の一覧です。これらが量子化の開始点と終了点となります。次の図に示すように、それらの間のメイングラフは、量子化可能でな場合のみ量子化されます。

図 22: TensorFlow の量子化フロー



X24607-091620

`-input_nodes` を前処理部の最後のノードに設定し、`-output_nodes` をメイングラフ部の最後のノードに設定することを推奨しています。つまり、前処理部と後処理部の一部の動作は量子化できないため、量子化したモデルを DPU で実行する必要がある場合に、Vitis AI クオンタイザーでコンパイルするとエラーが発生する可能性があります。

入力ノードは、グラフのプレースホルダー ノードとは異なる場合があります。凍結されたグラフにグラフ用の前処理部がない場合、プレースホルダー ノードは `input_nodes` に設定されている必要があります。

`input_fn` はプレースホルダー ノードと一致している必要があります。

[options] は、オプションのパラメーターを表します。最も一般的に使用されるオプションは次のとおりです。

- `weight_bit`: 量子化後の重みおよびバイアスのビット幅(デフォルトは 8)。
- `activation_bit`: 量子化後の活性化値のビット幅(デフォルトは 8)。
- `method`: 量子化方法。0 はオーバーフローなし、1 は最小差分法を表します。`non-overflow` は、量子化プロセスで値がオーバーフローしないようにします。量子化結果は、外れ値の影響を受けることがあります。`min-diffs` は、量子化プロセスでオーバーフローを許可し、量子化誤差を極力抑えることができます。外れ値に対する耐性があり、通常 `non-overflow` よりも範囲が狭くなります。

量子化されたモデルの出力

vai_q_tensorflow コマンドが正常に実行されると、\${output_dir} に 2 つのファイルが生成されます。

- quantize_eval_model.pb は、CPU/GPU で評価するために使用され、ハードウェアで結果をシミュレーションするために使用できます。tensorflow.contrib は遅延読み込みされるため、tensorflow.contrib.decent_q のインポートを明示的に実行して、カスタマイズした量子化動作をレジスターに記憶させる必要があります。
- deploy_model.pb を使用して DPU コードをコンパイルし、DPU 上でコードを運用します。これは Vitis AI コンパイラ用の入力ファイルとして使用できます。

表 8: vai_q_tensorflow 出力ファイル

No.	名前	説明
1	deploy_model.pb	DPUZDX8G インプリメンテーションをターゲットとする、VAI コンパイラ用の量子化されたモデル (拡張された Tensorflow 形式)。
2	quantize_eval_model.pb	評価用の量子化されたモデル (DPUCAHX8H、DPUCAHX8L、DPUCADF8H など、ほとんどの DPU アーキテクチャでは VAI コンパイラの入力)。

(オプション) 量子化されたモデルの評価

Vitis AI Model Zoo 内のモデルなどの浮動小数点モデルを評価するためのスクリプトがある場合は、次の 2 点を変更して、量子化されたモデルを評価できます。

- 浮動小数点モデル評価スクリプトの前に from tensorflow.contrib import decent_q を追加して、量子化操作を登録します。
- スクリプト内の浮動小数点モデルのパスを、量子化出力モデルに置き換えます。 "quantize_results/quantize_eval_model.pb"
- 変更したスクリプトを実行して、量子化されたモデルを評価します。

(オプション) シミュレーション結果のダンプ

量子化されたモデルの運用を開始すると、CPU/GPU のシミュレーション結果と DPU の出力値を比較することが必要になる場合があります。vai_q_tensorflow では、クオンタイザーで生成された quantize_eval_model.pb を使用したシミュレーション結果のダンプをサポートしています。

次のコマンドを実行して、量子化シミュレーション結果をダンプします。

```
$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
```

ダンプ用の input_fn は、量子化キャリブレーション用の input_fn と類似していますが、DPU の結果と一貫性を保つために、通常バッチサイズは 1 に設定されています。

上記のコマンドが正常に実行されると、 `${output_dir}` にダンプ結果が生成されます。 `${output_dir}` にはいくつかフォルダーがあり、各フォルダーには入力データのバッチのダンプ結果が含まれています。フォルダーには、各ノードの結果が個別に保存されます。量子化されたノードごとに、結果が `*_int8.bin` および `*_int8.txt` 形式で保存されます。dump_float が 1 に設定されている場合、量子化されていないノードの結果はダンプされます。記号「/」は、「_」に置き換えられてシンプルになります。ダンプ結果の例を次の表に示します。

表 9: ダンプ結果の例

バッチ番号	量子化	ノード名	保存されたファイル
1	あり	resnet_v1_50/conv1/biases/wquant	{output_dir}/dump_results_1/resnet_v1_50_conv1_biases_wquant_int8.bin {output_dir}/dump_results_1/resnet_v1_50_conv1_biases_wquant_int8.txt
2	なし	resnet_v1_50/conv1/biases	{output_dir}/dump_results_2/resnet_v1_50_conv1_biases.bin {output_dir}/dump_results_2/resnet_v1_50_conv1_biases.txt

vai_q_tensorflow 量子化微調整

量子化の微調整の手順は、浮動小数点モデルの微調整とほぼ同じです。ただし、量子化の微調整では、トレーニングを開始する前に `vai_q_tensorflow` の API を使用して浮動小数点グラフを書き換え、量子化されたグラフに変換します。一般的なワークフローは、次のとおりです。

1. 準備: 微調整の前に、次のファイルを準備してください。

表 10: `vai_q_tensorflow` 量子化微調整用の入力ファイル

No.	名前	説明
1	チェックポイント ファイル	出発点となる浮動小数点チェックポイント ファイル。ゼロからトレーニングする場合は省略できます。
2	データセット	ラベル付きのトレーニング データセット。
3	トレーニング スクリプト。	モデルの浮動小数点トレーニング/微調整を実行する Python スクリプト。

2. 浮動小数点モデルを評価する(オプション): 量子化の微調整を実行する前に、浮動小数点チェックポイント ファイルを評価し、スクリプトとデータセットに問題がないことを確認します。浮動小数点チェックポイントの精度とロス値は、量子化の微調整のベースラインとしても使用できます。
3. トレーニング スクリプトを変更する: 量子化トレーニング グラフを作成するには、トレーニング スクリプトを変更して、浮動小数点グラフの構築後に関数を呼び出します。次に例を示します。

```
# train.py
#
# Create the float training graph
model = model_fn(is_training=True)

# *Set the quantize configurations
from tensorflow.contrib import decent_q
q_config = decent_q.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
decent_q.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
# ...

```

QuantizeConfig には、量子化の設定が含まれます。

input_nodes、output_nodes、input_shapes などいくつかの基本設定は、モデル構造に従って指定する必要があります。

weight_bit、activation_bit、method などの他の設定にはデフォルト値があり、必要に応じて変更できます。すべての設定の詳細は、[vai_q_tensorflow の使用法](#) を参照してください。

- input_nodes/output_nodes: これらのパラメーターを使用して、量子化するサブグラフの範囲を指定します。前処理部および後処理部は通常は量子化できないため、この範囲から除外する必要があります。浮動小数点トレーニング グラフと浮動小数点評価グラフの量子化動作を完全に一致させるには、両方のグラフに同じ input_nodes と output_nodes を指定する必要があります。

注記: 現在、FIFO などの複数の出力テンソルを使用する量子化動作はサポートされていません。このような場合は、tf.identity ノードを追加して input_tensor のエイリアスを作成し、単一出力の入力ノードを作成してください。

- input_shapes: input_nodes の形状リスト。各ノードは 4 次元形状で、カンマで区切る必要があります (例: [[1,224,224,3] [1, 128, 128, 1]])。batch_size の不明なサイズをサポートします (例: [-1,224,224,3])。

4. 量子化されたモデルを評価し、運用モデルを生成する: 量子化の微調整後、運用モデルを生成します。その前に、チェックポイント ファイルを使用して量子化されたグラフを評価する必要があります。これを実行するには、浮動小数点評価グラフの作成後に次の関数を呼び出します。運用プロセスは量子化評価グラフに基づいて実行されるため、通常は量子化評価グラフを作成する関数と運用モデルを生成する関数と一緒に呼び出します。

```

# eval.py

# ...

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

# *Set the quantize configurations
from tensorflow.contrib import decent_q
q_config = decent_q.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
decent_q.CreateQuantizeEvaluationGraph(config=q_config)
# *Call Vai-q-tensorflow api to freeze the model and generate the deploy
model
decent_q.CreateQuantizeDeployGraph(checkpoint="path to checkpoint
folder", config=q_config)

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

```

生成されるファイル

上記の手順の実行後、次に示すファイルが \${output_dir} に生成されます。

表11: 生成済みファイルの情報

名前	TensorFlow との互換性	用途	説明
quantize_train_graph.pb	あり	トレーニング	量子化トレーニング グラフ。
quantize_eval_graph_{suffix}.pb	あり	チェックポイントによる評価	量子化情報が内部に凍結された量子化評価グラフ。内部に重みはありません。チェックポイントファイルと組み合わせて評価に使用されます。
quantize_eval_model_{suffix}.pb	あり	1.評価。2.ダンプ。 3.VAI コンパイラ (DPUCAHX8H)への入力	チェックポイント内の重みおよび量子化情報が内部に凍結された、凍結された量子化評価グラフ。このファイルを使用して、量子化されたモデルをホスト上で評価することや、各レイヤーの出力をダンプして DPU 出力とクロスチェックすることができます。XIR コンパイラはこのファイルを入力として使用します。
deploy_model_{suffix}.pb	なし	VAI コンパイラ (DPUCZDX8G)への入力	運用モデル、動作、および量子化の情報が融合されています。DNNC コンパイラはこのファイルを入力として使用します。

チェックポイントファイルとの組み合わせを明確にするために、接尾辞にチェックポイントファイルから得られるイテレーション情報と日付情報が含まれています。たとえば、チェックポイントファイルが「model.ckpt-2000.*」で日付が 20200611 の場合、接尾辞は「2000_20200611000000」になります。

TensorFlow 1.x 用の量子化微調整 API

一般的に量子化後の精度ロスはわずかとなりますが、Mobilenetなどの一部のネットワークでは、大きくなる可能性があります。このような場合、微調整を実行して量子化モデルの精度をさらに向上させることができます。

Python パッケージ `tf.contrib.decent_q` には、量子化微調整用の 3 つの API があります。

`tf.contrib.decent_q.CreateQuantizeTrainingGraph(config)`

デフォルト グラフ上でのインプレースの書き換えにより、浮動小数点トレーニング グラフを量子化トレーニング グラフに変換します。

引数

- config: 量子化の設定を含む `tf.contrib.decent_q.QuantizeConfig` オブジェクト。

`tf.contrib.decent_q.CreateQuantizeEvaluationGraph(config)`

デフォルト グラフ上でのインプレースの書き換えにより、浮動小数点評価グラフを量子化評価グラフに変換します。

引数

- config: 量子化の設定を含む `tf.contrib.decent_q.QuantizeConfig` オブジェクト。

`tf.contrib.decent_q.CreateQuantizeDeployGraph(checkpoint, config)`

チェックポイントを量子化評価グラフ内に凍結し、量子化評価グラフを運用グラフに変換します。

引数

- checkpoint: string オブジェクト。ファイルのチェックポイント フォルダーへのパス。

- config: 量子化の設定を含む `tf.contrib.decent_q.QuantizeConfig` オブジェクト。

量子化の微調整のヒント

次に量子化の微調整のヒントをいくつか示します。

- ドロップアウト:** 量子化の微調整は、ドロップアウト操作を使用しない方が効果が高いことが実験でわかっています。このツールは現在、ドロップアウトを使用した量子化の微調整をサポートしていないため、量子化の微調整を実行する前にドロップアウトを削除するか、または無効にする必要があります。ドロップアウトを無効にするには、`tf.layers` を使用する場合は `is_training=False` に設定し、`tf.keras.layers` を使用する場合は `tf.keras.backend.set_learning_phase(0)` を呼び出します。
- ハイパーパラメーター:** 量子化の微調整は浮動小数点モデルの微調整によく似ているため、浮動小数点モデルの微調整の手法も必要とされます。調整する必要がある重要なパラメーターとして、オプティマイザーのタイプ、学習曲線などが挙げられます。

vai_q_tensorflow でサポートされる動作および API

次の表に、vai_q_tensorflow でサポートされる動作および API を示します。

表 12: vai_q_tensorflow でサポートされる動作および API

タイプ	動作の種類	tf.nn	tf.layers	tf.keras.layers
Convolution	Conv2D DepthwiseConv2dNative	atrous_conv2d conv2d conv2d_transpose depthwise_conv2d_native separable_conv2d	Conv2D Conv2DTranspose SeparableConv2D	Conv2D Conv2DTranspose DepthwiseConv2D SeparableConv2D
Fully Connected	MatMul	/	Dense	Dense
BiasAdd	BiasAdd Add	bias_add	/	/
Pooling	AvgPool Mean MaxPool	avg_pool max_pool	AveragePooling2D MaxPooling2D	AveragePooling2D MaxPool2D
Activation	Relu Relu6 leaky_relu	relu relu6 leaky_relu	/	ReLU LeakyReLU
BatchNorm[#1]	FusedBatchNorm	batch_normalization batch_norm_with_global_normalization fused_batch_norm	BatchNormalization	BatchNormalization
Upsampling	ResizeBilinear ResizeNearestNeighbor	/	/	UpSampling2D
Concat	Concat ConcatV2	/	/	Concatenate

表12: **vai_q_tensorflow** でサポートされる動作およびAPI(続き)

タイプ	動作の種類	tf.nn	tf.layers	tf.keras.layers
その他	Placeholder Const Pad Squeeze Reshape ExpandDims	dropout[#2] softmax[#3]	Dropout[#2] Flatten	Input Flatten Reshape Zeropadding2D Softmax

注記:

- Conv2D/DepthwiseConv2D/Dense+BNのみをサポートします。推論を高速化するためにBNが含まれています。
- 推論を高速化するためにドロップアウトは削除されます。
- softmaxの出力を量子化する必要はないため、vai_q_tensorflowはこの量子化を実行しません。

vai_q_tensorflow の使用法

次の表に、vai_q_tensorflowでサポートされるオプションを示します。

 表13: **vai_q_tensorflow** のオプション

名前	タイプ	説明
共通コンフィギュレーション		
--input_frozen_graph	文字列	量子化キャリブレーションに使用される、浮動小数点モデル用のTensorFlowの凍結された推論 GraphDef ファイル。
--input_nodes	文字列	量子化グラフの入力ノード ネーム リストであり、「-output_nodes」と共に使用されます(カンマ区切り)。input_nodes と output_nodes は、量子化の始点/終了点です。それらの間のサブグラフは、量子化可能であれば量子化されます。 -input_nodes を前処理部の最後のノードに設定し、-output_nodes を後処理部の前の最後のノードに設定することを推奨しています。つまり、前処理部と後処理部の一部の動作は量子化できないため、量子化したモデルを DPU で実行する必要がある場合に、Vitis AI コンパイラでコンパイルするとエラーが発生する可能性があります。入力ノードは、グラフのプレースホルダー ノードとは異なる場合があります。
--output_nodes	文字列	量子化グラフの出力ノード ネーム リストであり、「-input_nodes」と共に使用されます(カンマ区切り)。入力ノードと出力ノードは、量子化の始点と終了点です。それらの間のサブグラフは、量子化可能であれば量子化されます。 -input_nodes を前処理部の最後のノードに設定し、-output_nodes を後処理部の前の最後のノードに設定することを推奨しています。つまり、前処理部と後処理部の一部の動作は量子化できないため、量子化したモデルを DPU で実行する必要がある場合に、Vitis AI コンパイラでコンパイルするとエラーが発生する可能性があります。
--input_shapes	文字列	input_nodes の形状リスト。各ノードは 4 次元形状で、カンマで区切られている必要があります(例: 1,224,224,3)。batch_size の不明なサイズをサポートします(例: ?,224,224,3)。複数の入力ノードの場合、各ノードの形状リストを「:」で区切って割り当てます(例: ?,224,224,3:?,300,300,1)。

表13: **vai_q_tensorflow** のオプション (続き)

名前	タイプ	説明
--input_fn	文字列	<p>この関数はグラフの入力データを提供し、キャリブレーションデータセットで使用されます。関数の形式は <code>module_name.input_fn_name</code> です (例: <code>my_input_fn.input_fn</code>)。<code>input_fn</code> は、キャリブレーションステップを示す入力として <code>int</code> オブジェクトを受け取り、呼び出しごとに <code>dict`{placeholder_node_name, numpy.Array}`</code> オブジェクトを返す必要があります。その後、モデルのプレースホルダー動作に渡されます。</p> <p>たとえば、<code>-input_fn</code> を <code>my_input_fn.calib_input</code> に割り当て、<code>my_input_fn.py</code> に <code>calib_input</code> 関数を次のように記述します。</p> <pre>def calib_input_fn: # read image and do some preprocessing return {"placeholder_1": input_1_narray, "placeholder_2": input_2_narray}</pre> <p>注記: 量子化中 <code>-input_nodes</code> の前のサブグラフが維持されているため、<code>input_fn</code> で再びグラフ用の前処理を実行する必要はありません。あらかじめ定義された入力関数 (デフォルトおよびランダムを含む) は、一般的には使用されないため削除してください。グラフ ファイルにない前処理部分は、<code>input_fn</code> で処理される必要があります。</p>
量子化の設定		
--weight_bit	Int32	量子化後の重みおよびバイアスのビット幅。 デフォルト: 8
--activation_bit	Int32	量子化後の活性化値のビット幅。 デフォルト: 8
--nodes_bit	文字列	ノードのビット幅を指定します。ノード名とビット幅はコロンで結合される1対のパラメーターを形成し、各パラメーターはカンマで区切られます。 <code>conv op</code> 名のみを指定すると、 <code>vai_q_tensorflow</code> は、指定されたビット幅を使用して <code>conv op</code> の重みを量子化します (例: 「conv1/Relu:16,conv1/weights:8,conv1:16」)。
--method	Int32	量子化の方法。 0: Non-overflow 量子化プロセスで値がオーバーフローしない。外れ値の影響を大きく受ける。 1: Min-diffs 量子化プロセスでオーバーフローを許可し、量子化誤差を極力抑える。外れ値に対する耐性がある。通常 non-overflow よりも範囲が狭くなる。 選択肢: [0, 1] デフォルト: 1
--nodes_method	文字列	ノードの手法を指定します。ノード名と手法はコロンで結合される1対のパラメーターを形成し、各パラメーターはカンマで区切られます。 <code>conv op</code> 名のみを指定すると、 <code>vai_q_tensorflow</code> は、指定された手法を使用して <code>conv op</code> の重みを量子化します (例: 「conv1/Relu:1,depthwise_conv1/weights:2,conv1:1」)。
calib_iter	Int32	キャリブレーションの反復。キャリブレーション用イメージの総数 = <code>calib_iter * batch_size</code> デフォルト: 100
--ignore_nodes	文字列	量子化の実行中に無視されるノードの名前リスト。無視されたノードは、量子化中に量子化されません。
--skip_check	Int32	1 に設定されている場合、浮動小数点モデルのチェックはスキップされます。これは入力モデルの一部のみが量子化される場合に有効です。 選択肢: [0, 1] デフォルト: 0

表13: `vai_q_tensorflow` のオプション(続き)

名前	タイプ	説明
<code>--align_concat</code>	Int32	連結ノードの入力の量子化位置を調整するための戦略。すべての連結ノードを整列させる場合は0に設定し、出力連結ノードを整列させる場合は1に設定し、整列を無効にするには2に設定します。 選択肢: [0, 1, 2] デフォルト: 0
<code>--simulate_dpu</code>	Int32	1に設定されている場合、DPUのシミュレーションが有効になります。一部の演算に対するDPUの動作はTensorFlowとは異なります。たとえば、LeakyReluおよびAvgPoolingの除算はビットシフトで置き換えられています。DPU出力とCPU/GPU出力にはわずかな違いが生じる可能性があります。このフラグが1に設定されている場合、 <code>vai_q_tensorflow</code> クォンタライザはこれらの動作をシミュレーションします。 選択肢: [0, 1] デフォルト: 1
<code>--adjust_shift_bias</code>	Int32	DPUコンパイラのシフトバイアスのチェックと調整のストラテジ。0に設定すると、シフトバイアスのチェックと調整を無効にします。1に設定すると、静的制約ありで有効にします。2に設定すると、動的制約ありで有効にします。 選択肢: [0, 1, 2] デフォルト: 1
<code>--adjust_shift_cut</code>	Int32	DPUコンパイラのシフトカットのチェックと調整のストラテジ。0に設定すると、シフトカットのチェックと調整を無効にします。1に設定すると、静的制約ありで有効にします。 選択肢: [0, 1] デフォルト: 1
<code>--arch_type</code>	文字列	固定ニューロンのアーキテクチャのタイプを指定します。「DEFAULT」では、重みとアクティベーションの両方の量子化範囲が[-128, 127]です。「DPUCADF8H」では、重みの量子化範囲が[-128, 127]で、アクティベーションの量子化範囲が[-127, 127]です。
<code>--output_dir</code>	文字列	量子化結果を保存するディレクトリ。 デフォルト: ./quantize_results
<code>--max_dump_batches</code>	Int32	ダンプ用の最大バッチ数。 デフォルト: 1
<code>--dump_float</code>	Int32	1に設定されている場合、浮動小数点の重みとアクティベーションもダンプされます。 選択肢: [0, 1] デフォルト: 0
<code>--dump_input_tensors</code>	文字列	グラフの入口がプレースホルダーではない場合、グラフの入力テンソル名を指定します。dump_input_tensorに従ってプレースホルダーを追加するため、input_fnはデータを供給できます。
セッションの設定		
<code>--gpu</code>	文字列	量子化に使用するGPUデバイスのID(カンマ区切り)。
<code>--gpu_memory_fraction</code>	Float	量子化に使用するGPUメモリの小数部(0~1)。 デフォルト: 0.5
その他		
<code>--help</code>		<code>vai_q_tensorflow</code> の有効なオプションをすべて表示します。
<code>--version</code>		<code>vai_q_tensorflow</code> のバージョン情報を表示します。

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
```

TensorFlow モデルのその他の量子化サンプルについては、[Xilinx Model Zoo](#) を参照してください。

TensorFlow 2.x バージョン (vai_q_tensorflow2)

vai_q_tensorflow2 のインストール

vai_q_tensorflow2 は次の 2 つの方法でインストールできます。

Docker コンテナーを使用してインストール

[Vitis AI](#) は、vai_q_tensorflow を含む量子化ツールの Docker コンテナーを提供します。コンテナーの実行後、Conda 環境 vitis-ai-tensorflow2 をアクティベートします。

```
conda activate vitis-ai-tensorflow2
```

ソース コードからインストール

vai_q_tensorflow2 は、[TensorFlow モデル最適化ツールキット](#) のフォークです。これは [Vitis_AI_Quantizer](#) 内のオープンソースです。vai_q_tensorflow2 を構築するには、次のコマンドを実行します。

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

vai_q_tensorflow2 の実行

次の手順に従って、vai_q_tensorflow を実行します。

浮動小数点モデルとキャリブレーション セットの準備

vai_q_tensorflow2 を実行する前に、次の表にリストされているファイルを含む浮動小数点モデルとキャリブレーション セットを準備します。

表14: **vai_q_tensorflow2**用の入力ファイル

No.	名前	説明
1	float model	TensorFlow2 浮動小数点モデル (h5 形式または保存されたモデルの形式)。
2	キャリブレーション データセット	入力データの分布を表現するトレーニング データセットまたは検証用データセットのサブセット。通常は 100 ~ 1000 個の画像で十分です。

vai_q_tensorflow2 API を使用した量子化

```
float_model = tf.keras.models.load_model('float_model.h5')
from tensorflow_model_optimization.quantization.keras import vitis_quantize
quantizer = vitis_quantize.VitisQuantizer(float_model)
quantized_model = quantizer.quantize_model(calib_dataset=calib_dataset)
```

ここでは、キャリブレーション用の代表的なキャリブレーション データセットの例として "calib_dataset" を使用します。eval_dataset、train_dataset、またはその他データセットのすべてあるいは一部を使用できます。train_dataset またはその他のデータセットを使用することもできます。クォンタイザーは、データセット全体を読み出してキャリブレーションします。tf.data.Dataset オブジェクトを使用する場合、バッチ サイズはデータセットによって制御されます。numpy.array オブジェクトを使用する場合、デフォルトのバッチ サイズは 50 です。

量子化されたモデルの保存

量子化されたモデル オブジェクトは、標準 `tf.keras` モデル オブジェクトです。これを保存するには、次のコマンドを実行します。

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

生成された `quantized_model.h5` ファイルは、`vai_c_tensorflow` コンパイラへ渡され、DPU で運用可能になります。

(オプション) 量子化されたモデルの評価

[Xilinx Model Zoo](#) のモデルなどの浮動小数点モデルを評価するためのスクリプトがある場合、評価のために、浮動小数点モデル ファイルを量子化されたモデルに置き換えることができます。カスタマイズされた量子化レイヤーをサポートするには、量子化されたモデルを「quantize_scope」にロードする必要があります。次に例を示します。

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

その後、浮動小数点モデルと同じように、量子化されたモデルを評価します。次に例を示します。

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

(オプション) シミュレーション結果のダンプ

量子化されたモデルの運用後に、CPU/GPU 上のシミュレーション結果と DPU 上の出力値を比較する必要が生じる場合があります。vai_q_tensorflow2 の VitisQuantizer.dump_model API を使用して、quantized_model でシミュレーション結果をダンプできます。

```
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,
    output_dir='./dump_results')
```

注記: DPU のデバッグの場合、dump_dataset の batch_size を 1 に設定します。

コマンドが正常に実行された後、ダンプ結果が \${dump_output_dir} に生成されます。各レイヤーの重みとアクティベーションの結果は、個別にフォルダーに保存されます。量子化されたレイヤーごとに、結果が *.bin および *.txt 形式で保存されます。レイヤーの出力が量子化されない場合は (softmax レイヤーの場合など)、浮動小数点モデルのアクティベーション結果が *_float.bin および *_float.txt 形式で保存されます。記号「/」は、「_」に置き換えられてシンプルになります。ダンプ結果の例を次の表に示します。

表 15: ダンプ結果の例

バッチ番号	量子化	レイヤー名	保存されたファイル		
			重み	バイアス	Activation
1	あり	resnet_v1_50/conv1	{output_dir}/dump_results_weights/quant_resnet_v1_50_conv1_kernel.bin {output_dir}/dump_results_weights/quant_resnet_v1_50_conv1_kernel.txt	{output_dir}/dump_results_weights/quant_resnet_v1_50_conv1_bias.bin {output_dir}/dump_results_weights/quant_resnet_v1_50_conv1_bias.txt	{output_dir}/dump_results_0/quant_resnet_v1_50_conv1.bin {output_dir}/dump_results_0/quant_resnet_v1_50_conv1.txt
2	なし	resnet_v1_50/softmax	N/A	N/A	{output_dir}/dump_results_0/quant_resnet_v1_50_softmax_float.bin {output_dir}/dump_results_0/quant_resnet_v1_50_softmax_float.txt

vai_q_tensorflow2 量子化微調整

一般的に量子化後の精度ロスはわずかとなりますが、MobileNet などの一部のネットワークでは、大きくなる可能性があります。このような場合、微調整を実行して量子化モデルの精度をさらに向上させることができます。

技術的には、量子化微調整は浮動小数点モデルの微調整とほぼ同じです。相違点は、トレーニングを開始する前に vai_q_tensorflow2 の API を使用して浮動小数点グラフを書き換え、量子化されたモデルに変換することです。一般的なワークフローは、次のとおりです。

1. 準備

微調整の前に、次のファイルを準備してください。

表 16: vai_q_tensorflow2 量子化微調整用の入力ファイル

No.	名前	説明
1	浮動小数点モデル ファイル	出発点となる浮動小数点モデル ファイル。ゼロからトレーニングする場合は省略できます。
2	データセット	ラベル付きのトレーニング データセット。
3	トレーニング スクリプト	モデルの浮動小数点トレーニング/微調整を実行する Python スクリプト。

2. (オプション) 浮動小数点モデルを評価する

量子化の微調整を実行する前に、浮動小数点モデルを評価し、スクリプトとデータセットに問題がないことを確認することを推奨します。浮動小数点チェックポイントの精度とロス値は、量子化の微調整のベースラインとしても使用できます。

3. トレーニング スクリプトを変更する

vai_q_tensorflow2 API (`VitisQuantizer.get_qat_model`) を使用して、量子化を実行します。次に例を示します。

```
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)
model = quantizer.get_qat_model()

# Compile the model
model.compile(
    optimizer= RMSprop(learning_rate=lr_schedule),
    loss=tf.keras.losses.SparseCategoricalCrossentropy(),
    metrics=keras.metrics.SparseTopKCategoricalAccuracy())

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

4. モデルを保存する

`model.save()` を呼び出してトレーニング済みモデルを保存するか、または `model.fit()` 内のコールバックを使用してモデルを定期的に保存します。次に例を示します。

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

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

5. (オプション) 量子化されたモデルを評価する

`eval_dataset` 上で `model.evaluate()` を呼び出し、浮動小数点モデルの評価と同じように、量子化されたモデルを評価します。

注記: 量子化の微調整の動作は浮動小数点モデルの微調整に類似しているため、浮動小数点モデルのトレーニングと微調整の経験を積むには非常に有益です。たとえば、オプティマイザーのタイプや学習レートなどのハイパーパラメーターをどのように選択するかを学習できます。

vai_q_tensorflow2 でサポートされる動作および API

次の表に、vai_q_tensorflow2 でサポートされる動作および API を示します。

表 17: vai_q_tensorflow2 でサポートされるレイヤー

サポートされるレイヤー
tf.keras.layers.Conv2D
tf.keras.layers.Conv2DTranspose
tf.keras.layers.DepthwiseConv2D
tf.keras.layers.Dense
tf.keras.layers.AveragePooling2D
tf.keras.layers.MaxPooling2D
tf.keras.layers.GlobalAveragePooling
tf.keras.layers.UpSampling2D
tf.keras.layers.BatchNormalization
tf.keras.layers.Concatenate
tf.keras.layers.Zeropadding2D
tf.keras.layers.Flatten
tf.keras.layers.Reshape
tf.keras.layers.ReLU
tf.keras.layers.Activation
tf.keras.layers.Add

vai_q_tensorflow2 の用途

```
vitis_quantize.VitisQuantizer(model, custom_quantize_strategy=None)
```

クラス VitisQuantizer の構築関数。

引数:

- model: 量子化の設定を含む tf.keras.Model オブジェクト。
- custom_quantize_strategy: カスタム量子化ストラテジ json ファイル。

```
vitis_quantize.VitisQuantizer.quantize_model(
calib_dataset=None,
fold_conv_bn=True,
fold_bn=True,
replace_relu6=True,
include_cle=True,
cle_steps=10)
```

この関数は、浮動小数点モデルを量子化します。これには、モデルの最適化、重みの量子化、およびアクティベーションの量子化キャリブレーションが含まれます。

引数:

- `calib_dataset`: `tf.data.Dataset` または `np.numpy` オブジェクト。キャリブレーションデータセット。
- `fold_conv_bn`: `bool` オブジェクト。バッチ正規化レイヤーを前の `Conv2D/DepthwiseConv2D/TransposeConv2D/Dense` レイヤーに折りたたむかどうか。
- `fold_bn`: `bool` オブジェクト。バッチ正規化レイヤーの `moving_mean` および `moving_variance` 値をガンマ値およびベータ値に折りたたむかどうか。
- `replace_relu6`: `bool` オブジェクト。`Relu6` レイヤーを `Relu` レイヤーに置き換えるかどうか。
- `include_cle`: `bool` オブジェクト。量子化の前にクロス レイヤー イコライゼーションを実行するかどうか。
- `cle_steps`: `int` オブジェクト。クロス レイヤー イコライゼーションを実行するための反復手順。

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

この関数は、量子化されたモデルのシミュレーション結果をダンプします。これには、モデルの最適化、重みの量子化、およびアクティベーションの量子化キャリブレーションが含まれます。

引数:

- `model`: 量子化の設定を含む `tf.keras.Model` オブジェクト。
- `dataset`: `tf.data.Dataset` または `np.numpy` オブジェクト。ダンプに使用されるデータセット。`weights_only` が `True` に設定されている場合は不要です。
- `output_dir`: `string` オブジェクト。ダンプ結果を保存するディレクトリ。
- `weights_only`: `bool` オブジェクト。`True` に設定すると、重みのみをダンプします。`False` に設定すると、アクティベーション結果もダンプします。

例

- 量子化する:

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

- 量子化されたモデルを評価する:

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

- 量子化されたモデルを読み込む:

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

- 量子化されたモデルをダンプする:

```
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)
```

Pytorch バージョン (vai_q_pytorch)

vai_q_pytorch のインストール

vai_q_pytorch には、GPU バージョンと CPU バージョンがあります。vai_q_pytorch は、PyTorch バージョン 1.2 ~ 1.4 をサポートしますが、PyTorch のデータ並列処理はサポートしません。vai_q_pytorch をインストールする方法は 2 つあります。

Docker コンテナーを使用してインストール

Vitis AI は、vai_q_pytorch を含む量子化ツールの Docker コンテナーを提供します。GPU/CPU コンテナーの実行後、Conda 環境 vitis-ai-pytorch をアクティベートします。

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

注記: Conda 環境に複数のパッケージをインストールする場合、パーミッションの問題をクリアしていれば、vitis-ai-pytorch を直接使用する代わりに、vitis-ai-pytorch をベースとして個別の Conda 環境を作成できる場合があります。Xilinx Model Zoo の pt_pointpillars_kitti_12000_100_10.8G_1.3 モデルはこの例です。

ソース コードからインストール

vai_q_pytorch は、Pytorch プラグインとして機能するように設計された Python パッケージです。これは Vitis AI Quantizer 内のオープンソースです。vai_q_pytorch は Conda 環境にインストールすることを推奨します。これを行うには、次の手順に従います。

- .bashrc に CUDA_HOME 環境変数を追加します。

GPU バージョンでは、CUDA ライブラリが /usr/local/cuda にインストールされている場合、.bashrc に次の行を追加します。CUDA がその他のディレクトリにある場合は、それに応じてパスを変更します。

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

CPU バージョンでは、.bashrc 内のすべての CUDA_HOME 環境変数設定を削除します。次のコマンドを実行して、シェル ウィンドウのコマンド ラインで削除することを推奨します。

```
unset CUDA_HOME
```

2. PyTorch (1.2 ~ 1.4) と torchvision をインストールします。

次のコードでは、PyTorch 1.4 と torchvision 0.5.0 を例として使用します。その他のバージョンの手順の詳細は、[PyTorch のウェブサイト](#)を参照してください。

```
pip install torch==1.4.0 torchvision==0.5.0
```

3. その他の依存ライブラリをインストールします。

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

4. vai_q_pytorch をインストールします。

```
cd ./pytorch_binding  
python setup.py install (for user)  
python setup.py develop (for developer)
```

5. インストールを検証します。

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

注記: バージョン 1.4 より古い PyTorch をインストールした場合は、スクリプトに `torch` をインポートする前に `pytorch_nndct` をインポートしてください。これはバージョン 1.4 以前の Pytorch のバグによるものです。詳細は、PyTorch の GitHub の issue [28536](#) および [19668](#) を参照してください。

```
import pytorch_nndct  
import torch
```

vai_q_pytorch の実行

vai_q_pytorch は、Pytorch プラグインとして機能するように設計されています。ザイリンクスは、FPGA に最適な量子化機能を実装するために、シンプルな API を提供しています。明確に定義されたモデルでは、わずか 2 ~ 3 行を追加するだけで量子化モデル オブジェクトを作成できます。これを行うには、次の手順に従います。

vai_q_pytorch 用のファイルを準備する

vai_q_pytorch 用に次のファイルを準備します。

表 18: **vai_q_pytorch** 用の入力ファイル

No.	名前	説明
1	model.pth	トレーニング済みの Pytorch モデル (通常は .pth ファイル)。
2	model.py	浮動小数点モデルの定義を含む Python スクリプト。
3	キャリブレーション データ セット	100 ~ 1000 個のイメージを含むトレーニング データセットの一部。

モデルの定義を変更する

Pytorch モデルの量子化を可能にするには、モデルの定義を変更して次の条件を満たすようにする必要があります。サンプルは [Vitis AI の Github](#) にあります。

1. 量子化されるモデルには、forward メソッドのみが含まれている必要があります。その他のすべての関数は、外部または派生クラスへ移動する必要があります。これらの関数は、通常は前処理および後処理として機能します。これらの関数は、外部へ移動しなければ API によって量子化されるモジュール内で削除されるため、量子化されたモジュールの転送時に予測不可能な動作が発生する可能性があります。
2. 浮動小数点モデルは、jit トレース テストに合格する必要があります。浮動小数点モジュールを評価ステータスに設定し、`torch.jit.trace` 関数を使用して浮動小数点モデルをテストします。

vai_q_pytorch API を浮動小数点スクリプトに追加する

量子化の前に、トレーニング済みの浮動小数点モデルと、モデルの精度/mAP を評価するための Python スクリプトがある場合、クオンタイザー API は浮動小数点モジュールを量子化済みモジュールに置き換えます。通常の評価関数では、量子化済みモジュールの転送が推奨されます。フラグ `quant_mode` が「calib」に設定されている場合、量子化キャリブレーションが評価プロセス中のテンソルの量子化ステップを決定します。キャリブレーションの実行後、`quant_mode` を「test」に設定して、量子化されたモデルを評価します。

1. `vai_q_pytorch` モジュールをインポートします。

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

2. 量子化に必要な入力でクオンタイザーを生成し、変換されたモデルを取得します。

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

3. 変換されたモデルと共にニューラルネットワークを転送します。

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

4. 量子化の結果を出力し、モデルを運用します。

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

量子化を実行し、結果を取得する

注記: `vai_q_pytorch` ログ メッセージは、専用の色と特殊なキーワード「NNDCT」で示されます。「NNDCT」は内部プロジェクト名で、後から変更できます。`vai_q_pytorch` ログ メッセージには、「error」、「warning」、「note」の3種類があります。`vai_q_pytorch` ログ メッセージに注意して、フロー ステータスを確認してください。

1. 「--quant_mode calib」を指定してコマンドを実行し、モデルを量子化します。

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

キャリブレーションを転送する際は、浮動小数点モデルの評価フローを借用して、浮動小数点スクリプトからのコードの変更を最小限に抑えます。最後にロスと精度に関するメッセージが表示された場合は、無視してかまいません。特殊なキーワード「NNDCT」が付いた色付きのログ メッセージに注意してください。

量子化中および評価中の反復回数を制御することは重要です。一般的に、量子化には 100 ~ 1000 個の画像があれば十分であり、評価には検証セット全体が必要とされます。反復回数は、データ読み込み部で制御できます。この場合、引数「subset_len」により、ネットワーク転送に何個の画像を使用するかを制御できます。浮動小数点モデルの評価スクリプトに同じ役割を持つ引数が含まれていない場合は、引数を追加するか、手動で変更する必要があります。

この量子化コマンドが正常に実行されると、出力ディレクトリ "./quantize_result" の下に 2 つの重要なファイルが生成されます。

```
ResNet.py: converted vai_q_pytorch format model,
Quant_info.json: quantization steps of tensors got. (Keep it for
evaluation of quantized model)
```

2. 次のコマンドを実行して、量子化されたモデルを評価します。

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

コマンドが正常に実行された後に表示される精度は、量子化されたモデルの正確な精度です。

3. コンパイル用の xmodel を生成するには、バッチ サイズを 1 に設定する必要があります。冗長な反復を避けるために subset_len=1 に設定し、次のコマンドを実行します。

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

実行中にログに表示されるロスと精度は無視します。Vitis AI コンパイラ用の xmodel ファイルが、出力ディレクトリ ./quantize_result に生成されます。このファイルは FPGA での運用に使用されます。

```
ResNet_int.xmodel: deployed model
```

注記: XIR は Vitis AI Docker の「vitis-ai-pytorch」Conda 環境で使用可能になりますが、vai_q_pytorch をソースコードからインストールする場合は、前もって XIR をインストールしておく必要があります。XIR がインストールされていない場合、xmodel ファイルは生成されず、コマンドはエラーを返します。この場合も、出力ログで精度をチェックすることは可能です。

モジュールの部分的量子化

モデル内のすべてのサブモジュールを量子化する必要がない場合は、モジュールの部分的量子化を使用できます。汎用 vai_q_pytorch API に加えて QuantStub/DeQuantStub 演算子ペアを使用することで、これを実現できます。次の例は、subm0 および subm2 を量子化し、subm1 を量子化しない方法を示しています。

```
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
```

vai_q_pytorch 高速微調整

一般的に量子化後の精度ロスはわずかとなります。MobileNetなどの一部のネットワークでは、大きくなる可能性があります。このような状況では、まず最初に高速微調整を試します。高速微調整を実行しても満足のいく結果が得られない場合は、量子化微調整を使用して、量子化モデルの精度をさらに向上させることができます。

AdaQuant アルゴリズム¹は、ラベルなしのデータの小さな集合を使用して、アクティベーションを調整するだけでなく、重みも微調整します。AdaQuant は、ラベルなしのデータの小さな集合を使用します。これはキャリブレーションと類似していますが、モデルを微調整します。Vitis AI クオンタイザーが実装するこのアルゴリズムは、「高速微調整」または「高度なキャリブレーション」と呼ばれます。高速微調整は、量子化キャリブレーションより多少低速ですが、より優れた性能を達成できます。高速微調整は量子化微調整と類似していますが、実行するたびに異なる結果が生成されます。

高速微調整はモデルの実際のトレーニングではないため、必要な反復回数は限定されます。Imagenet データセットの分類モデルの場合、1000 個の画像で十分です。高速微調整には、モデル評価スクリプトに基づいた若干の変更が必要ですが、トレーニングのためにオプティマイザーを設定する必要はありません。高速微調整を使用するには、モデル転送を繰り返すための関数が必要であり、これは高速微調整を実行中に呼び出されます。元の推論コードを使用して再キャリブレーションすることを強く推奨します。

サンプル全体は [オープンソース サンプル](#) にあります。

```
# fast finetune model or load finetuned parameter before test
if fast_finetune == True:
    ft_loader, _ = load_data(
        subset_len=1024,
        train=False,
        batch_size=batch_size,
        sample_method=None,
        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()
```

パラメーターの微調整と、この ResNet18 サンプルの再キャリブレーションを実行するには、次のコマンドを実行します。

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

微調整された量子化済みモデルの精度をテストするには、次のコマンドを実行します。

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

注記:

1. Itay Hubara et.al., Improving Post Training Neural Quantization: Layer-wise Calibration and Integer Programming, arXiv:2006.10518, 2020.

vai_q_pytorch 量子化微調整

事前に定義されたモデルアーキテクチャがある場合は、次の手順に従って QAT(量子化認識トレーニング)を実行します。ここでは、例として torchvision の ResNet18 モデルを使用します。モデル全体の定義は、[こちら](#) を参照してください。

1. 量子化される非モジュール動作があるかどうかをチェックします。

ResNet18は、「+」を使用して2つのテンソルを加算します。これらを `pytorch_nnndct.nn.modules.functional.Add` に置き換えます。

2. 複数回呼び出されるモジュールがあるかどうかをチェックします。

通常はこのようなモジュールには重みがありません。最も一般的なモジュールは `torch.nn.ReLU` モジュールです。このような複数のモジュールを定義し、往方向で個別に読み出します。必要条件を満たす、修正した定義は次のようにになります。

```
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 '+'
        return self.skip_add(out, identity)
```

```
# out += identity
    out = self.skip_add(out, identity)
    out = self.relu2(out)

    return out
```

3. QuantStub および DeQuantStub を挿入します。

QuantStub を使用してネットワークの入力を量子化し、DeQuantStub を使用してネットワークの出力を逆量子化します。往方向で QuantStub から DeQuantStub までのすべてのサブネットワークが量子化されます。QuantStub/DeQuantStub のペアは複数指定できます。

```
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 is None:
            # 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. 量子化微調整 API を使用してクォンタイザーを作成し、モデルをトレーニングします。

```

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
quantizer = torch_quantizer(quant_mode = 'calib',
                            module = model,
                            input_args = input,
                            bitwidth = 8,
                            qat_proc = True)
quantized_model = quantizer.quant_model
optimizer = torch.optim.Adam(
    quantized_model.parameters(), lr, weight_decay=weight_decay)

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

```

5. トレーニング済みモデルを運用可能なモデルに変換します。

トレーニングの実行後、量子化されたモデルを xmodel にダンプします (xmodel のコンパイルには、batch_size=1 の指定が必須)。

```

# vail_q_pytorch interface function: deploy the trained model and convert
xmodel
    # need at least 1 iteration of inference with batch_size=1
    quantizer.deploy(quantized_model)
    deployable_model = quantizer.deploy_model
    val_dataset2 = torch.utils.data.Subset(val_dataset, list(range(1)))
    val_loader2 = torch.utils.data.DataLoader(
        val_dataset,
        batch_size=1,
        shuffle=False,
        num_workers=workers,
        pin_memory=True)
    validate(val_loader2, deployable_model, criterion, gpu)
    quantizer.export_xmodel()

```

vail_q_pytorch 量子化微調整の必要条件

一般的に量子化後の精度ロスはわずかとなりますが、MobileNetなどの一部のネットワークでは、大きくなる可能性があります。このような状況では、まず最初に高速微調整を試します。高速微調整を実行しても満足のいく結果が得られない場合は、量子化微調整によって量子化モデルの精度をさらに向上させることができます。

量子化微調整 API には、モデルのトレーニングのための必要条件がいくつかあります。

- 量子化されるすべての動作は、torch 関数や Python 演算子ではなく、`torch.nn.Module` オブジェクトのインスタンスでなければなりません。たとえば、PyTorch では一般的に ‘+’ を使用して 2 つのテンソルを加算しますが、この演算子は量子化微調整ではサポートされません。したがって、‘+’ を `pytorch_nndct.nn.modules.functional.Add` に置き換えます。次の表に、置き換えが必要な動作のリストを示します。

表 19: 動作の置き換えマップ

動作	置換対象
+	<code>pytorch_nndct.nn.modules.functional.Add</code>
-	<code>pytorch_nndct.nn.modules.functional.Sub</code>

表 19: 動作の置き換えマップ(続き)

動作	置換対象
<code>torch.add</code>	<code>pytorch_nndct.nn.modules.functional.Add</code>
<code>torch.sub</code>	<code>pytorch_nndct.nn.modules.functional.Sub</code>



重要: 量子化されるモジュールをフォワード パス内で複数回呼び出すことはできません。

2. 量子化されるネットワークの始めと終わりに、`pytorch_nndct.nn.QuantStub` および `pytorch_nndct.nn.DeQuantStub` を使用します。ネットワークは、ネットワーク全体でも部分的なサブネットワークでもかまいません。

vai_q_pytorch の使用法

このセクションでは、量子化を実装し、ターゲット ハードウェア上で運用されるモデルを生成するための、実行ツールと API の使用法を説明します。モジュール `pytorch_binding/pytorch_nndct/apis/quant_api.py` 内の API は次のとおりです。

```
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"),
                 qat_proc: bool = False):
```

クラス `torch_quantizer` は、クオントライザ オブジェクトを作成します。

引数:

- `quant_mode`: プロセスが使用している量子化モードを指定する整数。「calib」は量子化のキャリブレーション、「test」は量子化されたモデルの評価。
- `module`: 量子化される浮動小数点モジュール。
- `input_args`: 量子化される浮動小数点モジュールの実際の入力と同じ形状を持つ入力テンソル。値は乱数にすることができます。
- `state_dict_file`: 浮動小数点モジュールのトレーニング済みパラメーター ファイル。浮動小数点モジュールにパラメーターが読み込まれている場合は、このパラメーターを設定する必要はありません。
- `output_dir`: 量子化の結果と中間ファイルを格納するディレクトリ。デフォルトは「quantize_result」です。
- `bitwidth`: グローバルな量子化ビット幅。デフォルトは 8 です。
- `device`: GPU または CPU 上の実行モデル。
- `qat_proc`: 量子化微調整をオンにします。量子化微調整は QAT (量子化認識トレーニング) とも呼ばれます。

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

この関数は、量子化ステップの情報をエクスポートします。

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

この関数は、xmodel をエクスポートし、詳細なデータ比較用に演算子の出力データをダンプします。

引数:

- `output_dir`: 量子化の結果と中間ファイルを格納するディレクトリ。デフォルトは「quantize_result」です。
- `deploy_check`: 詳細なデータ比較用のデータのダンプを制御するフラグ。デフォルトは `False` です。`True` に設定されている場合、バイナリ形式のデータが `output_dir/deploy_check_data_int/` にダンプされます。

Caffe バージョン (`vai_q_caffe`)

`vai_q_caffe` のインストール

`vai_q_caffe` をインストールする方法は 2 つあります。

Docker コンテナーを使用してインストール

Vitis AI は、`vai_q_caffe` を含む量子化ツールの Docker コンテナーを提供します。コンテナーの実行後、Conda 環境 `vitis-ai-caffe` をアクティベートします。

```
conda activate vitis-ai-caffe
```

ソース コードからインストール

`vai_q_caffe` は、[caffe_xilinx](#) リポジトリ内のオープンソースです。`vai_q_caffe` は、[NVIDIA Caffe](#) のブランチ「`caffe-0.15`」からのザイリンクスが保守管理するフォークです。構築プロセスは、[BVLC Caffe](#) と同じです。[こちら](#) のインストール手順を参照してください。

`vai_q_caffe` の実行

次の手順に従って、`vai_q_caffe` を実行します。

1. ニューラル ネットワーク モデルを準備する

表 20: `vai_q_caffe` の入力ファイル

No.	名前	説明
1	float.prototxt	ResNet-50 の浮動小数点モデル。prototxt のデータレイヤーは、キャリブレーション データセットのパスと一致している必要があります。
2	float.caffemodel	ResNet-50 用のトレーニング済み重みファイル。
3	キャリブレーション データセット	学習用画像のサブセット (画像数 100 ~ 1000)。

`vai_q_caffe` を実行する前に、次に示すキャリブレーション データセットを含む浮動小数点形式の Caffe モデルを準備します。

- Caffe 浮動小数点ネットワーク モデルの prototxt ファイル。
- 学習済み Caffe 浮動小数点ネットワーク モデルの caffemodel ファイル。
- キャリブレーション データ セット。通常は、学習用の画像または実際のアプリケーションで用いる画像のサブセット(100 以上)をキャリブレーション セットとして使用します。image_data_param の source と root_folder を実際のキャリブレーション イメージ リストとイメージ フォルダーパスに設定する必要があります。量子化キャリブレーションには、ラベルなしのキャリブレーション データで十分ですが、実装のため、2 列のイメージ リスト ファイルが必要になります。したがって、2 つ目の列にはランダム値または 0 を設定してください。次に calibration.txt の例を示します。

```
n01440764_985.JPG 0
n01443537_9347.JPG 0
n01484850_8799.JPG 0
...
```

図 23: 量子化に使用する Caffe レイヤーの例

```
# ResNet-50
name: "ResNet-50"
layer {
    name: "data"
    type: "ImageData"
    top: "data"
    top: "label"
    include {
        phase: TRAIN
    }
    transform_param {
        mirror: false
        mean_value: 104
        mean_value: 107
        mean_value: 123
    }
    image_data_param {
        source: "./data/imagenet_256/calibration.txt"
        root_folder: "./data/imagenet_256/calibration_images/"
        batch_size: 10
        shuffle: false
        new_height: 224
        new_width: 224
    }
}
```

注記: チャネルの 3 つの mean_value パラメーターが推奨されています。3 つの mean_value パラメーターを指定する場合は、BGR 順序に従ってください。

- vai_q_caffe を実行して量子化されたモデルを生成します。

```
vai_q_caffe quantize -model float.prototxt -weights float.caffemodel
[options]
```

ハードウェア プラットフォームにはわずかな違いがあるため、vai_q_caffe の出力形式も異なります。ターゲット ハードウェア プラットフォームが DPUCAHX8H の場合、コマンドに -keep_fixed_neuron オプションを追加する必要があります。

```
vai_q_caffe quantize -model float.prototxt -weights float.caffemodel -
keep_fixed_neuron[options]
```

3. 上記のコマンドが正常に実行されると、4つのファイルが出力ディレクトリ (デフォルト ディレクトリ: ./quantize_results/) に生成されます。deploy.prototxt および deploy.caffemodel ファイルは、コンパイラへの入力ファイルとして使用されます。quantize_train_test.prototxt および quantize_train_test.caffemodel ファイルは、GPU/CPU の精度をテストするために使用され、微調整を量子化するための入力ファイルとして使用できます。

表 21: **vai_q_caffe** 出力ファイル

No.	名前	説明
1	deploy.prototxt	Vitis AI コンパイラ用の量子化されたネットワーク記述ファイル。
2	deploy.caffemodel	Vitis AI コンパイラ用の量子化された Caffe モデルのパラメーター ファイル (非標準 Caffe 形式)。
3	quantize_train_test.prototxt	テストおよび微調整用の量子化されたネットワーク記述ファイル。
4	quantize_train_test.caffemodel	テストおよび微調整用の量子化された Caffe モデルのパラメーター ファイル (非標準 Caffe 形式)。

量子化されたモデルの精度を評価するには、次のようなコマンドを使用するか、または手順 2 で -auto_test 「-auto_test」を追加します。**vai_q_caffe** の引数の詳細は、次のセクションを参照してください。

```
vai_q_caffe test -model ./quantize_results/quantize_train_test.prototxt -weights ./quantize_results/quantize_train_test.caffemodel -gpu 0 -test_iter 1000
```

vai_q_caffe 量子化微調整

一般的に量子化後の精度ロスはわずかとなりますが、Mobilenetなどの一部のネットワークでは、大きくなる可能性があります。このようなシナリオでは、微調整を実行して量子化モデルの精度をさらに向上させることができます。

微調整はモデルトレーニングとほぼ同じで、オリジナルのトレーニングデータセットと solver.prototxt が必要です。quantize_train_test.prototxt および Caffe モデルで微調整を始めるには、次の手順に従います。

1. quantize_train_test.prototxt の入力レイヤーにトレーニングデータセットを割り当てます。
2. 微調整用の solver.prototxt ファイルを作成します。solver.prototxt ファイルの例を次に示します。良い結果を得るために、ハイパーパラメーターを調整することができます。最も重要なパラメーターは base_lr です。これは通常、トレーニングで使用されるパラメーターよりも相当小さくなります。

```

net: "./fix_results/fix_train_test.prototxt"
test_iter: 2500
test_interval: 2000
test_INITIALIZATION: false
display: 10
average_loss: 100
base_lr: 0.0000001
lr_policy: "poly"
power: 1
gamma: 0.1
max_iter: 2000
momentum: 0.9
weight_decay: 0.0000
snapshot: 1000
snapshot_prefix: "./finetune/"
snapshot_diff: false
solver_mode: GPU
iter_size: 1

```

3. 次のコマンドを実行して、微調整を開始します。

```
./vai_q_caffe finetune -solver solver.prototxt -weights quantize_results/
quantize_train_test.caffemodel -gpu all
```

4. 微調整済みモデルをデプロイします。微調整されたモデルは、\${snapshot_prefix}/finetuned_iter10000.caffemodel などの solver.prototxt ファイルの snapshot_prefix 設定で生成されます。test コマンドを使用すると、精度をテストできます。
5. 最後に、deploy コマンドを使用して、Vitis AI コンパイラ用のデプロイ モデル (prototxt および caffemodel) を生成できます。

```
./vai_q_caffe deploy -model quantize_results/
quantize_train_test.prototxt -weights finetuned_iter10000.caffemodel -
gpu 0 -output_dir deploy_output
```

vai_q_caffe の使用法

vai_q_caffe クオンタイザーは、浮動小数点モデルを入力モデルとして使用し、キャリブレーション データセットを使用して量子化されたモデルを生成します。次のコマンド ラインの [options] はオプションのパラメーターを表しています。

```
vai_q_caffe quantize -model float.prototxt -weights float.caffemodel
[options]
```

次の表に、vai_q_caffe でサポートされているオプションを示します。最も一般的に使用される 3 つのオプションは、weights_bit、data_bit、method です。

表 22: **vai_q_caffe** オプションの一覧

名前	タイプ	オプション	デフォルト	説明
model	文字列	必須	-	浮動小数点 prototxt ファイル (float.prototxt など)。
weights	文字列	必須	-	トレーニング済みの浮動小数点の重み (float.caffemodel など)。
weights_bit	Int32	オプション	8	量子化後の重みおよびバイアスのビット幅。

表 22: **vai_q_caffe** オプションの一覧 (続き)

名前	タイプ	オプション	デフォルト	説明
data_bit	Int32	オプション	8	量子化後の活性化値のビット幅。
method	Int32	オプション	1	量子化方法。0はオーバーフローなし、1は最小差分法を表します。 non-overflowは、量子化プロセスで値がオーバーフローしないようにします。また、外れ値の影響を大きく受けます。 min-diffsは、量子化プロセスでオーバーフローを許可し、量子化誤差を極力抑えることができます。外れ値に対する耐性があり、通常non-overflowよりも範囲が狭くなります。
calib_iter	Int32	オプション	100	キャリブレーションの最大の繰り返し回数。
auto_test	-	オプション	未使用	このオプションを追加すると、キャリブレーション後に prototxt ファイルで指定したテストデータセットを使用してテストが実行されます。このオプションをオンにすることは、浮動小数点 prototxt ファイルが TRAIN モードと TEST モードの両方で実行可能な精度計算用 prototxt になっている必要があります。
test_iter	Int32	オプション	50	テストの最大の繰り返し回数。
output_dir	文字列	オプション	quantize_results	量子化結果用の出力ディレクトリ。
gpu	文字列	オプション	0	キャリブレーションおよびテスト用の GPU デバイス ID。
ignore_layers	文字列	オプション	none	量子化の際に無視するレイヤーのリスト。
ignore_layers_file	文字列	オプション	none	量子化中に無視するレイヤーを定義する Protobuf ファイル (ignore_layers で始まる)。
sigmoided_layers	文字列	オプション	none	sigmoid 演算前のレイヤー リスト。精度の最適化で量子化されます。
input_blob	文字列	オプション	data	blob 入力データ。
keep_fixed_neuron	ブール	オプション	FALSE(オフ)	運用しているモデルの FixedNeuron レイヤーを維持します。このフラグは、ターゲットハードウェア プラットフォームが DPUCAHX8H の場合に設定します。

例:

- 量子化する:

```
vai_q_caffe quantize -model float.prototxt -weights float.caffemodel -gpu
0
```

- 自動テストで量子化する:

```
vai_q_caffe quantize -model float.prototxt -weights float.caffemodel -gpu
0 -auto_test -test_iter 50
```

- オーバーフロー以外の手法で量子化する:

```
vai_q_caffe quantize -model float.prototxt -weights float.caffemodel -gpu 0 -method 0
```

- 量子化されたモデルを微調整する:

```
vai_q_caffe finetune -solver solver.prototxt -weights quantize_results/float_train_test.caffemodel -gpu 0
```

- 量子化されたモデルを運用する:

```
vai_q_caffe deploy -model quantize_results/quantize_train_test.prototxt -weights quantize_results/float_train_test.caffemodel -gpu 0
```

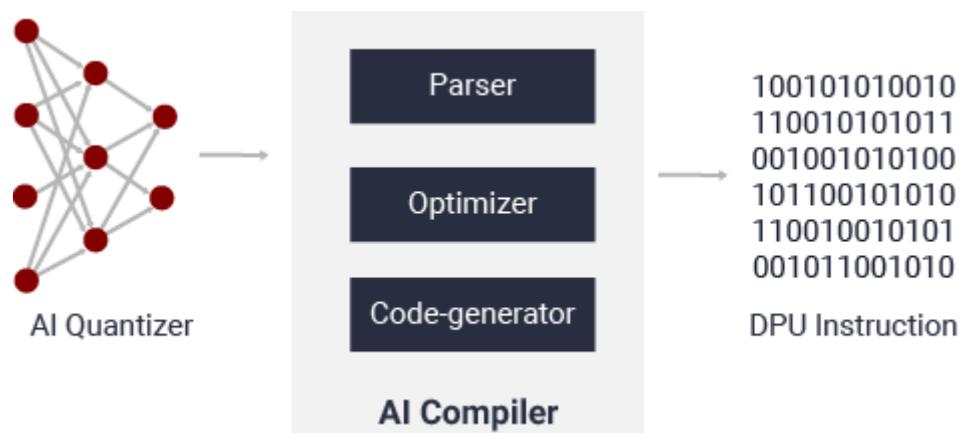
モデルのコンパイル

Vitis AI コンパイラ

Vitis™ AI コンパイラ (VAI_C) は、各種の DPU に対するニューラル ネットワーク計算の最適化に使用されるコンパイラ ファミリへの統合インターフェイスです。各コンパイラは、ネットワーク モデルを、高度に最適化された DPU 命令シーケンスにマップします。

次の図に、VAI_C フレームワークを簡単に示します。最適化および量子化された入力モデルのトポロジを解析した後、VAI_C は内部計算グラフ（中間表現、IR）を構築し、対応する制御フローとデータフローを提供します。その後、複数の最適化を実行します。たとえば、バッチ正規化が先行するたたみ込みに融合される計算ノードの融合や、本来備わっている並列処理を利用した効率的な命令スケジューリング、またはデータの再利用などがあります。

図 24: Vitis AI コンパイラのフレームワーク



Vitis AI コンパイラは、DPU のマイクロアーキテクチャに基づいてコンパイルされたモデルを生成します。Vitis AI では、さまざまなプラットフォームとアプリケーションに対応する数多くの DPU がサポートされます。使用可能なコンパイラとそれに関連する DPU の関係を理解することが重要です。DPU の命名規則については、[DPU 名](#) を参照してください。

コンパイラと DPU の対応関係をよく理解するために、次の表を参照してください。

表 23: コンパイラと DPU の対応関係

DPU 名	コンパイラ	ハードウェア プラットフォーム
DPUCZDX8G	XCompiler	Zynq UltraScale+ MPSoC、Zynq-7000 デバイス
DPUCAHX8H		U50、U280
DPUCAHX8L		U50、U280
DPUCADF8H		U200、U250
DPUCVDX8G		VCK190、Versal AI コア シリーズ
DPUCVDX8H		VCK5000
DPUCADX8G	xfDNN コンパイラ	U200、U250

XCompiler は XIR ベースのコンパイラです。このコンパイラは、DPUCZDX8G、DPUCAHX8H、DPUCAHX8L、DPUCVDX8G、および DPUCVDX8H をサポートします。xfDNN コンパイラは、レガシ ML Suite に由来するコンパイラで、DPUCADX8G のみをサポートします。このコンパイラは Vitis AI 1.3 リリースでは下位互換性の維持のために残されていますが、Vitis AI 1.4 リリースでは廃止される予定です。

XIR ベースのツールチェーンによるコンパイル

ザイリンクス XIR (Intermediate Representation) は、AI アルゴリズムのグラフ ベースの中間表現で、FPGA プラットフォーム上での DPU のコンパイルと効率的な運用のために設計されています。アドバンス ユーザーは、Vitis AI フローで XIR を拡張してカスタマイズ IP をサポートすることにより、FPGA の能力を最大限に引き出し、アプリケーション全体を高速化できます。XIR は現在、Vitis AI クオンタイザー、コンパイラ、ランタイムおよびその他のツールの基盤となっています。

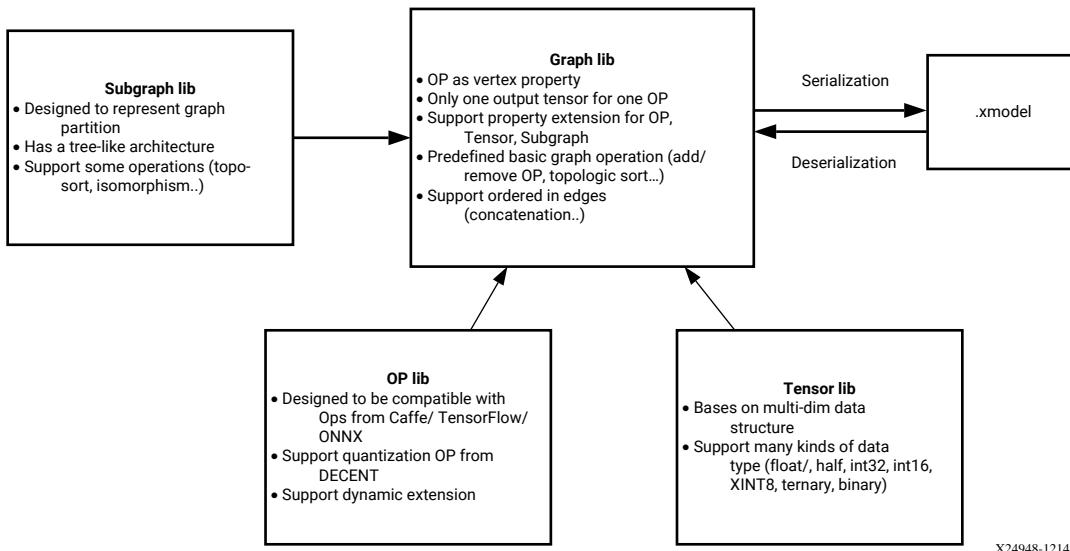
XIR

XIR には、Op、テンソル、グラフ、およびサブグラフの各ライブラリが含まれており、計算グラフを明確かつ柔軟に表現できます。XIR には、異なる用途向けにインメモリ形式とファイル形式があります。インメモリ形式の XIR はグラフ オブジェクトで、ファイル形式の XIR は xmodel です。グラフ オブジェクトは xmodel にシリアル化でき、xmodel はグラフ オブジェクトにデシリアル化できます。

Op ライブラリには、広く使用されている深層学習フレームワーク (TensorFlow、PyTorch、Caffe) に対応する明確に定義された演算子のセットと、すべての内蔵 DPU 演算子があります。これによって表現能力が強化され、これらのフレームワーク間の違いを解消してユーザーと開発者に統一的な表現を提供するという主要な目的の 1 つが達成されます。

また XIR は、PyXIR という名前の Python API を提供します。この API により、Python ユーザーは Python 環境で XIR を利用できます。つまり、異なる言語間のギャップを埋めるために大量の作業をしなくても、Python プロジェクトと現在の XIR ベースのツールを統合して協調開発が可能となります。

図25: XIRベースフロー



X24948-I21420

xir::Graph

グラフは XIR の主要なコンポーネントです。グラフは複数の重要な API (`xir::Graph::serialize`, `xir::Graph::deserialize`、および `xir::Graph::topological_sort` など) を取得します。

グラフはコンテナーに似ており、Op を頂点として保持し、生産者/消費者の関係をエッジとして使用します。

xir::Op

XIR 内の Op は、XIR に内蔵または XIR から拡張された、演算子定義のインスタンスです。すべての Op インスタンスは、事前に定義された内蔵/拡張された Op 定義ライブラリに従って、グラフによって作成または追加されます。Op の定義には、主に入力引数とイントリニシクス属性が含まれます。

Op インスタンスは、事前に定義されたイントリニシクス属性以外に、`xir::Op::set_attr` API を適用することにより、より多くのエクストリニシクス属性を提供できます。各 Op インスタンスは出力テンソルを 1 つだけ取得でき、ファンアウト Op は 2 つ以上取得できます。

xir::Tensor

テンソルも XIR の重要なクラスです。ほかのフレームワークのテンソル定義とは異なり、XIR のテンソルは、そのテンソルが表現するデータ ブロックの記述です。実際のデータ ブロックは、テンソルから除外されます。

テンソルの主要な属性は、データ型と形状です。

xir::Subgraph

XIR のサブグラフは、Op の集合をオーバーラップしない複数の集合に分割したツリー状の階層です。グラフの Op セット全体は、ルートと見なすことができます。サブグラフは入れ子にできますが、オーバーラップしないことが必要です。入れ子にされた内側のサブグラフは、外側のサブグラフの子である必要があります。

DPU 向けのコンパイル

XIR ベース コンパイラは、量子化された TensorFlow または Caffe モデルを入力として受け取ります。最初に、このプロセスの土台として、入力モデルを XIR 形式に変換します。異なるフレームワーク間におけるほぼすべての違いが排除され、XIR という統一表現に変換されます。その後、グラフ上でさまざまな最適化を適用し、OP を DPU で実行できるかどうかに基づいて、グラフを複数のサブグラフに分割します。必要に応じて、各サブグラフに対して、よりアーキテクチャを意識した最適化が適用されます。DPU サブグラフ用にコンパイラが命令ストリームを生成し、それをサブグラフに付け加えます。最後に、VART 用に必要な情報と命令を含む最適化済みのグラフが、コンパイル済みの xmodel ファイルへシリアル化されます。

XIR ベースのコンパイラは、ZCU エッジ プラットフォーム上では DPUCZDX8G シリーズ、高スループット アプリケーションに最適化された Alveo HBM プラットフォーム上では DPUCAHX8H、低レイテンシ アプリケーションに最適化された Alveo HBM プラットフォーム上では DPUCAHX8L、Versal エッジ プラットフォーム上では DPUCVDX8G、Versal クラウド プラットフォーム上では DPUCVDX8H をサポートします。これらのプラットフォーム用の arch.json ファイルは、/opt/vitis_ai/compiler/arch にあります。

VAI_C で Caffe または TensorFlow モデルをコンパイルする手順は、前述した DPU と同じです。この場合、VAI_C を含む Vitis AI パッケージが正常にインストールされており、vai_quantizer でモデルが圧縮されていることが前提となります。

Caffe

Caffe では、vai_q_caffe が PROTOTXT (deploy.prototxt) と MODEL (deploy.caffemodel) を生成します。vai_q_caffe には、XIR ベースのコンパイラに必須の -keep_fixed_neuron オプションを必ず指定してください。次のコマンドを実行して、コンパイル済みの xmodel を取得します。

```
vai_c_caffe -p /PATH/TO/deploy.prototxt -c /PATH/TO/deploy.caffemodel -a /  
PATH/TO/arch.json -o /OUTPUTPATH -n netname
```

コンパイラは、OUTPUTPATH ディレクトリに 3 つのファイルを作成します。netname_org.xmodel は、コンパイラのフロントエンドで生成されるコンパイル済みの xmodel です。netname.xmodel は、命令やその他の必要情報を含むコンパイル済みの xmodel です。meta.json はランタイム用です。

TensorFlow

TensorFlow では、vai_q_tensorflow が pb ファイル (quantize_eval_model.pb) を生成します。vai_q_tensorflow によって生成される pb ファイルは 2 つあります。quantize_eval_model.pb ファイルは、XIR ベースのコンパイラ用の入力ファイルです。コンパイル コマンドは同じです。

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

出力は Caffe の出力と同じです。

TensorFlow モデルには入力テンソルの形状情報が含まれない場合があり、この場合はコンパイルは失敗します。--options '{"input_shape": "1,224,224,3"}' のような追加オプションを指定して、入力テンソルの形状を指定できます。

TensorFlow 2.x

TensorFlow 2.x では、クオンタイザーが量子化済みモデルを hdf5 形式で生成します。

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

現在、vai_c_tensorflow2 は Keras 機能 API のみをサポートします。シーケンシャル API は、今後のリリースでサポートされる予定です。

PyTorch

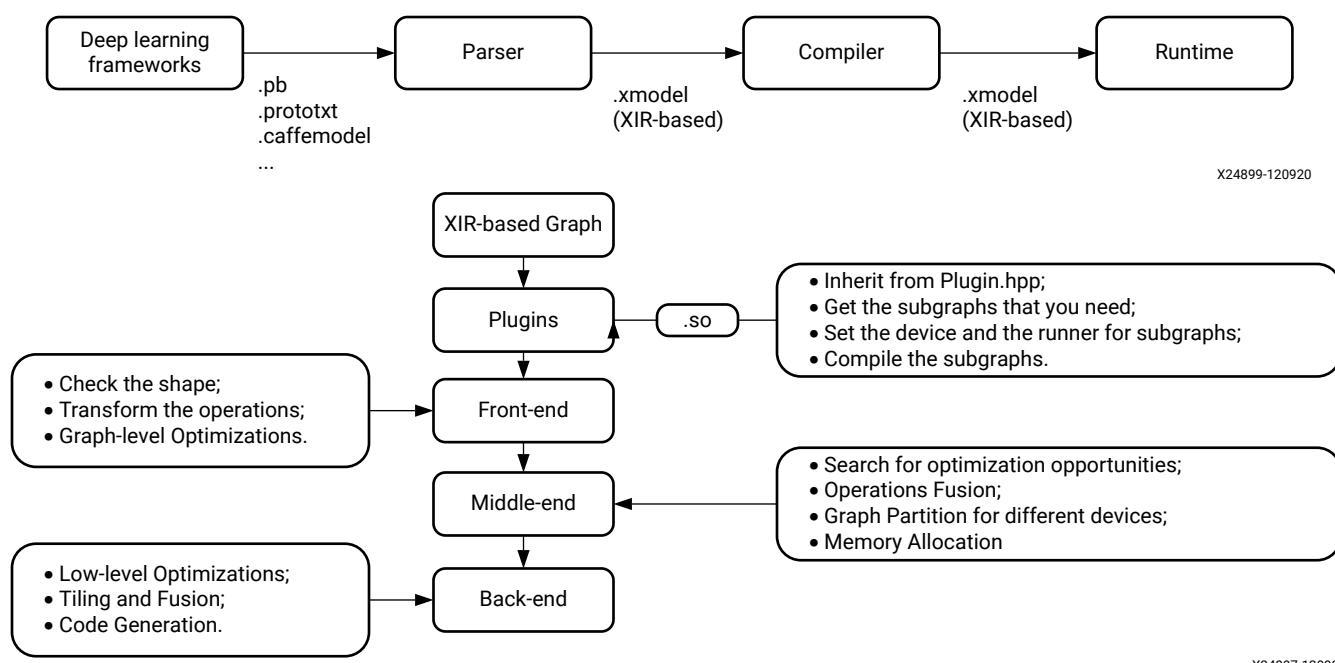
PyTorch では、クオンタイザー NNDCT が量子化済みモデルを XIR 形式で直接出力します。vai_c_xir を使用してモデルをコンパイルします。

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

カスタマイズされたアクセラレータ用のコンパイル

XIR ベースのコンパイラは、深層学習フレームワークから生成される、フレームワークに依存しない XIR グラフのコンテキスト内で動作します。パーサーは CNN モデルからフレームワーク固有の属性を削除し、モデルを XIR ベースの計算グラフに変換します。コンパイラは、計算グラフをさまざまなサブグラフに分割し、ヘテロジニアスな最適化を利用して、各サブグラフに対応する最適化されたマシンコードを生成します。

図 26: コンパイル フロー



DPU でサポートされない op がモデルに含まれている場合は、いくつかのサブグラフが作成されて CPU にマップされます。FPGA は処理能力が高いため、特定の IP を作成してこれらの op を高速化し、エンドツーエンドのパフォーマンスを向上させることができます。XIR ベースのツールチェーンを使用して、カスタマイズされたアクセラレーション IP を有効にするには、プラグインと呼ばれるパイプラインを利用して XIR とコンパイラを拡張します。

インターフェイス クラスのプラグインは、`Plugin.hpp` 内で宣言されます。コンパイラが DPU 用のグラフのコンパイルを開始する前に、プラグインが順番に実行されます。最初に、演算子ごとに子サブグラフが作成され、高速化できる演算子がプラグインによって選択されます。次に、これらのサブグラフは大きなサブグラフに結合され、カスタマイズ IP にマップされて、ランタイム (VART::Runner) 用の必要な情報 (サブグラフに関する指示など) が追加されます。

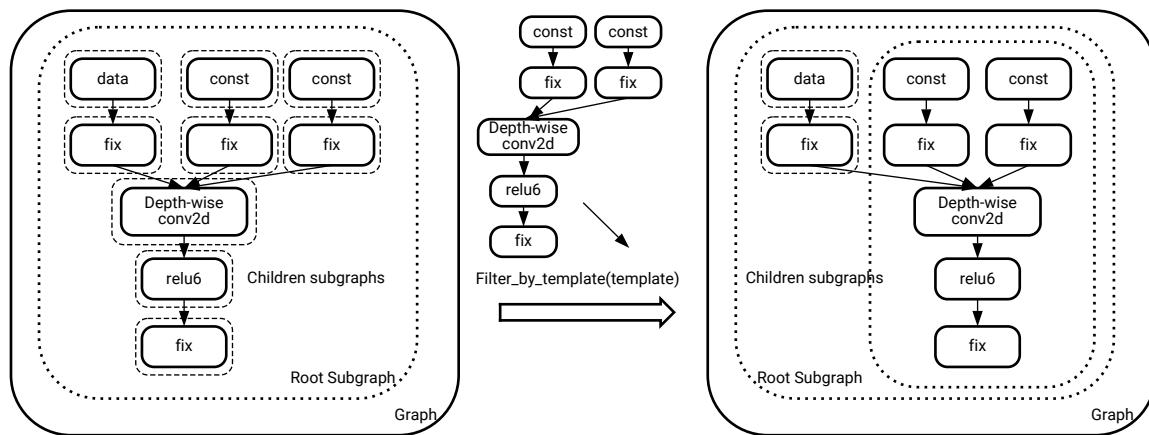
プラグインの実装

- Plugin::partition() を実装します。

`std::set<xir::Subgraph*> partition(xir::Graph* graph)` で、目的の op を選択し、デバイス レベルのサブグラフに結合します。次のヘルパー関数を使用できます。

- `xir::Subgraph* filter_by_name(xir::Graph* graph, const std::string& name)` は、特定の名前を持つサブグラフを返します。
- `std::set<xir::Subgraph*> filter_by_type(xir::Graph* graph, const std::string& type)` は、特定のタイプの詳細サブグラフを返します。
- `std::set<xir::Subgraph*> filter_by_template(xir::Graph* graph, xir::GraphTemplate* temp)` は、特定の構造を持つサブグラフを返します。

図 27: テンプレートによるフィルタリング



X24895-121520

- `std::set<xir::Subgraph*> filter(xir::Graph* graph, std::function<std::set<xir::Subgraph*>(<std::set<xir::Subgraph*>>) func)` により、カスタマイズ関数によってサブグラフをフィルタリングできます。この手法は、コンパイルされていないサブグラフをすべて見つけるのに役立ちます。

取得した子サブグラフを結合する必要がある場合は、`merge_subgraph()` という名前のヘルパー関数を使用して子サブグラフを結合します。ただし、この関数が結合できるのは、同じレベルのサブグラフのみです。サブグラフのリストを 1 つのサブグラフに結合できない場合、ヘルパー関数は可能な限りそのリストを結合します。

- Plugin::partition() 関数で選択したサブグラフの名前、デバイス、およびランナーを指定します。
- Plugin::compile(xir::Subgraph*) を実装します。この関数は、partition() 関数によって返されるすべてのサブグラフに対して呼び出されます。ここで希望する動作を実行できます。また、ランタイム用のサブグラフに情報を追加できます。

プラグインの構築

外部 `get_plugin()` 関数を作成し、共有ライブラリ内にインプリメンテーションを構築する必要があります。

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

プラグインの使用

vai_c コマンド ライン オプションで `--options '{"plugin": "libplugin0.so,libplugin1.so"}'` を使用して、プラグイン ライブラリをコンパイラに渡すことができます。プラグインの実行中に、コンパイラはライブラリを開き、「get_plugin」という名前の外部関数をロードすることにより、プラグインのインスタンスを作成します。2つ以上のプラグインを指定した場合、プラグインはコマンド ライン オプションで指定した順番で実行されます。すべてのプラグインが実装された後、DPU および CPU 用のコンパイルが実行されます。

サポートされている OP と DPU の制限

現在サポートされている演算子

ザイリンクスは、DPU IP およびコンパイラを引き続き改善し、より優れた性能を持つ多くの演算子をサポートします。次の表に、DPU でサポートされる標準的な動作と設定(カーネル サイズ、ストライドなど)を示します。動作の設定がこれらの制限値を超えると、その演算子は CPU に割り当てられます。DPU でサポートされる演算子は、DPU のタイプ、ISA バージョン、および設定によって異なります。

DPU が各種の FPGA デバイスに適応できるように、一部の DPU はコンフィギュレーション可能になっています。必要なエンジンの選択、イントリニシクス パラメーターの調整、独自の DPU IP と TRD プロジェクトの作成が可能です。DPU がコンフィギュレーション可能であることは、制限値がコンフィギュレーションによって大きく異なることを意味します。これらのオプションが制限値にどのように影響を与えるかの詳細は、PG338 を参照してください。ユーザー独自の DPU 設定でモデルをコンパイルしてみることを推奨します。コンパイラは、どの演算子が CPU に割り当てられるかとその理由を通知します。次の表は、各 DPU アーキテクチャの特定のコンフィギュレーションを示しています。

表 24: 現在サポートされている演算子

CNN での標準的な動作のタイプ	パラメーター	DPUCZDX8G_ISA0_B4096_MAX_BG2 (ZCU102/104)	DPUCAHX8L_ISA0 (U280)	DPUCAHX8H_ISA2 (U50LV9E、U50LV10E、U280)、 PUCAHX8H_ISA2_ELP2 (U50)	DPUCVDX8G_ISA0_B8192C 32B3 (VCK190)	DPUCVDX8H_ISA0 (VCK5000)
イントリニシクス パラメーター		channel_parallel: 16 bank_depth: 2048	channel_parallel: 32 bank_depth: 4096	channel_parallel: 16 bank_depth: 2048	channel_parallel: 16 bank_depth: 16384	channel_parallel: 64 bank_depth: 256
conv2d	Kernel size	w、h: [1, 16]	w、h: [1, 16]	w、h: [1, 16]	w、h: [1, 16] w * h <= 64	w、h: [1, 16]
	Strides	w、h: [1, 8]	w、h: [1, 4]	w、h: [1, 4]	w、h: [1, 4]	w、h: [1, 4]
	Dilation	dilation * input_channel <= 256 * channel_parallel				
	Paddings	pad_left、pad_right: [0, (kernel_w - 1) * dilation_w + 1]				
		pad_top、pad_bottom: [0, (kernel_h - 1) * dilation_h + 1]				
	In Size	kernel_w * kernel_h * ceil(input_channel / channel_parallel) <= bank_depth				
	Out Size	output_channel <= 256 * channel_parallel				
	Activation	ReLU、LeakyReLU、ReLU6	ReLU、ReLU6	ReLU、LeakyReLU、ReLU6	ReLU、LeakyReLU、ReLU6	ReLU、LeakyReLU
	Group* (Caffe)	group==1				

表24: 現在サポートされている演算子(続き)

CNNでの標準的な動作のタイプ	パラメーター	DPUCZDX8G_ISA0_B4096_MAX_BG2 (ZCU102/104)	DPUCAHX8L_ISA0 (U280)	DPUCAHX8H_ISA2 (U50LV9E、U50LV10E、U280)、PUCAHX8H_ISA2_ELP2 (U50)	DPUCVDX8G_ISA0_B8192C 32B3 (VCK190)	DPUCVDX8H_ISA0 (VCK5000)										
イントリンシクス パラメーター	channel_parallel: 16 bank_depth: 2048	channel_parallel: 32 bank_depth: 4096	channel_parallel: 16 bank_depth: 2048	channel_parallel: 16 bank_depth: 16384	channel_parallel: 64 bank_depth: 256											
depthwise-conv2d	Kernel size	w、h: [1, 16]	w, h: [3]	サポートされない												
	Strides	w、h: [1, 8]	w, h: [1, 2]													
	dilation	dilation * input_channel <= 256 * channel_parallel														
	Paddings	pad_left、pad_right: [0, (kernel_w - 1) * dilation_w + 1]	pad_top、pad_bottom: [0, (kernel_h - 1) * dilation_h + 1]													
		pad_top、pad_bottom: [0, (kernel_h - 1) * dilation_h + 1]														
	In Size	kernel_w * kernel_h * ceil(input_channel / channel_parallel) <= bank_depth														
	Out Size	output_channel <= 256 * channel_parallel														
	Activation	ReLU、ReLU6	ReLU、ReLU6													
	Group* (Caffe)	group==input_channel														
transposed-conv2d	Kernel size	kernel_w/stride_w, kernel_h/stride_h: [1, 16]														
	Strides	pad_left、pad_right: [1, kernel_w-1]														
	Paddings	pad_top、pad_bottom: [1, kernel_h-1]														
		output_channel <= 256 * channel_parallel														
	Activation	ReLU、LeakyReLU、ReLU6	ReLU、ReLU6	ReLU、LeakyReLU、ReLU6	ReLU、LeakyReLU、ReLU6	ReLU、LeakyReLU										
depthwise-transposed-conv2d	Kernel size	kernel_w/stride_w, kernel_h/stride_h: [1, 16]	kernel_w/stride_w, kernel_h/stride_h: [3]	サポートされない												
	Strides	pad_left、pad_right: [1, kernel_w-1]														
	Paddings	pad_top、pad_bottom: [1, kernel_h-1]														
		output_channel <= 256 * channel_parallel														
	Activation	ReLU、ReLU6	ReLU、ReLU6													

表24: 現在サポートされている演算子(続き)

CNNでの標準的な動作のタイプ	パラメーター	DPUCZDX8G_ISA0_B4096_MAX_BG2 (ZCU102/104)	DPUCAHX8L_ISA0 (U280)	DPUCAHX8H_ISA2 (U50LV9E、U50LV10E、U280)、PUCAHX8H_ISA2_ELP2 (U50)	DPUCVDX8G_ISA0_B8192C 32B3 (VCK190)	DPUCVDX8H_ISA0 (VCK5000)					
イントリンシクス パラメーター	channel_parallel: 16 bank_depth: 2048	channel_parallel: 32 bank_depth: 4096	channel_parallel: 16 bank_depth: 2048	channel_parallel: 16 bank_depth: 16384	channel_parallel: 64 bank_depth: 256						
max-pooling	Kernel size	w、h: [2, 8]	w, h: {2, 3, 5, 7, 8}	w, h: [1, 8]	w, h: [2, 8]	w, h: {1, 2, 3, 7}					
	Strides	w, h: [1, 8]	w, h: [1, 8]	w, h: [1, 8]	w, h: [1, 4]	w, h: [1, 8]					
	Paddings	pad_left、pad_right: [1, kernel_w-1] pad_top、pad_bottom: [1, kernel_h-1]									
	Activation	ReLU	サポートされない	ReLU	ReLU	サポートされない					
average-pooling	Kernel size	w、h: [2, 8] w==h	w, h: {2, 3, 5, 7, 8} w==h	w, h: [1, 8] w==h	w, h: [2, 8] w==h	w, h: {1, 2, 3, 7} w==h					
	Strides	w, h: [1, 8]	w, h: [1, 8]	w, h: [1, 8]	w, h: [1, 4]	w, h: [1, 8]					
	Paddings	pad_left、pad_right: [1, kernel_w-1] pad_top、pad_bottom: [1, kernel_h-1]									
	Activation	ReLU	サポートされない	ReLU	ReLU	サポートされない					
eltwise-sum	Input Channel	input_channel <= 256 * channel_parallel									
	Activation	ReLU	ReLU	ReLU	ReLU	ReLU					
concat	ネットワーク固有の制限。機能マップのサイズ、量子化の結果およびコンパイラの最適化に関連します。										
reorg	Strides	reverse==false : stride ^ 2 * input_channel <= 256 * channel_parallel reverse==true : input_channel <= 256 * channel_parallel									
pad	In Size	input_channel <= 256 * channel_parallel									
	Mode	「SYMMETRIC」(「CONSTANT」パッドはコンパイラの最適化プロセス中に隣接する演算子に融合される)									
global pooling	グローバル プーリングは、入力テンソル サイズに等しいカーネル サイズで一般的なプーリングとして処理されます。										
InnerProduct、Fully Connected、Matmul	これらの演算子は、1x1 に等しいカーネル サイズで conv2d op に変換されます。										

次に、さまざまな深層学習フレームワークで従来から使用されている演算子を示します。コンパイラは、これらの演算子を自動的に解析し、XIR 形式に変換し、DPU または CPU に分配できます。これらの演算子は、部分的にツールでサポートされており、参照用としてここに一覧を示します。

TensorFlowでサポートされる演算子

表25: TensorFlowでサポートされる演算子

TensorFlow		XIR		DPUのインプリメンテーション
OPの種類	属性	OP名	属性	
placeholder / inputlayer*	shape	data	shape	入力データ用のメモリを割り当てます。
			data_type	
const		const	data_shape data_type	constデータ用のメモリを割り当てます。
conv2d	filter	conv2d	kernel	たたみ込みエンジン
	strides		stride	
			pad([0, 0, 0, 0])	
	padding		pad_mode(SAME または VALID)	
	dilations		dilation	
depthwiseconv2dnative	filter	depthwise-conv2d	kernel	depthwise たたみ込みエンジン
	strides		stride	
	explicit_paddings		padding	
	padding		pad_mode(SAME または VALID)	
	dilations		dilation	
conv2dbackpropinput / conv2dtranspose*	filter	transposed-conv2d	kernel	たたみ込みエンジン
	strides		stride	
			padding([0, 0, 0, 0])	
	padding		pad_mode(SAME または VALID)	
	dilations		dilation	
spacetobacthnd + conv2d + batchtospacend	block_shape	conv2d	dilation	設定した特定の要件を満たす場合、Spacetobatch、Conv2dおよびBatchtospaceはたたみ込みエンジンにマップされます。
	padding			
	filter		kernel	
	strides		stride	
	padding		pad_mode(SAME)	
	dilations		dilations	
	block_shape			
matmul / dense*	transpose_a	conv2d / matmul	transpose_a	等価な conv2d がハードウェア要件を満たし、DPU にマップできる場合、matmul は conv2d 動作に変換されます。
	transpose_b		transpose_b	
maxpool / maxpooling2d*	ksize	maxpool	kernel	ブーリング エンジン
	strides		stride	
			pad([0, 0, 0, 0])	
	padding		pad_mode(SAME または VALID)	

表25: TensorFlowでサポートされる演算子(続き)

TensorFlow		XIR		DPUのインプリメンテーション
OPの種類	属性	OP名	属性	
avgpool / averagepooling2d* / globalaveragemeanpooling2d*	ksize	avgpool	kernel	ブーリングエンジン
	strides		stride	
			pad([0, 0, 0, 0])	
	padding		pad_mode(SAME または VALID)	
			count_include_pad(false)	
			count_include_invalid(true)	
mean	axis	avgpool / reduction_mean	axis	等価な avgpool がハードウェア要件を満たし、DPUにマップできる場合、mean動作は avgpool に変換されます。
	keep_dims		keep_dims	
relu		relu		アクティベーションは、convolution、addなどの隣接する動作に融合されます。
relu6		relu6		
leakyrelu	alpha	leakyrelu	alpha	
fixneuron / quantizelayer*	bit_width	fix	bit_width	コンパイル中に float2fix と fix2float に分割され、float2fix および fix2float の動作は隣接する動作と融合されて粗粒度の動作に変換されます。
	quantize_pos		fix_point	
			if_signed	
			round_mode	
identity		identity		identity は削除されます。
add、addv2		add		add が要素ごとの加算である場合、add は DPU の要素ごとの加算エンジンにマップされます。add がチャネルごとの加算である場合、add とたたみ込みなどの隣接する動作を融合できる可能性を探します。
concatv2 / concatenate*	axis	concat	axis	特殊な読み出しままたは書き込みストラテジとオンチップメモリの慎重な割り当てにより、concat の結果生じるオーバーヘッドを低減します。
pad / zeropadding2d*	paddings	pad	paddings	「CONSTANT」パディングは、隣接する動作と融合されます。 「SYMMETRIC」パディングは、DPU命令にマップされます。「REFLECT」パディングは、まだ DPUによってサポートされていません。
	mode		mode	

表25: TensorFlowでサポートされる演算子(続き)

TensorFlow		XIR		DPUのインプリメンテーション
OPの種類	属性	OP名	属性	
shape		shape		shape動作は削除されます。
stridedslice	begin	stridedslice	begin	shapeに関連する動作である場合、コンパイル中に削除されます。粗粒度の動作のコンボネントである場合、隣接する動作と融合されます。それ以外の場合、CPUインプリメンテーションにコンパイルされます。
	end		end	
	strides		strides	
pack	axis	stack	axis	
neg		neg		
mul		mul		
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	
resizebilinear	size/scale	resize	size	resizeのモードが「BILINEAR」、align_corner=false、half_pixel_centers = false、size = 2、4、8である場合、align_corner=false、half_pixel_centers = false、size = 2、4は、DPUインプリメンテーション(pad +depthwise-転置されたconv2d)に変換できます。
	align_corners		align_corners	
	half_pixel_centers		half_pixel_centers	
			mode="BILINEAR"	
resizenearestneighbor	size/scale	resize	size	align_corner=false、half_pixel_centers = true、size = 2、4は、DPUインプリメンテーション(pad +depthwise-転置されたconv2d)に変換できます。resizeのモードが「NEAREST」でsizeが整数である場合、resizeはDPUインプリメンテーションにマップされます。
	align_corners		align_corners	
	half_pixel_centers		half_pixel_centers	
			mode="NEAREST"	
upsample2d	size/scale	resize	size	resize動作に変換される場合があります。それ以外の場合、CPUインプリメンテーションにマップされます。
			align_corners	
			half_pixel_centers	
	interpolation		mode	
reshape	shape	reshape	shape	
transpose	perm	transpose	order	
squeeze	axis	squeeze	axis	
exp		exp		CPUインプリメンテーションにのみコンパイルされます。
softmax	axis	softmax	axis	
sigmoid		sigmoid		

表 25: TensorFlow でサポートされる演算子 (続き)

TensorFlow		XIR		DPU のインプリメンテーション
OP の種類	属性	OP 名	属性	
square+rsqrt+maximum		l2_normalize	axis	output = $x / \sqrt{\max(\sum(x^2), \epsilon)}$ は、XIR で l2_normalize に融合されます。
			epsilon	

注記:

- 上記の TensorFlow の OP は、XIR でサポートされます。これらのすべての OP には、ツールチェーン内に CPU インプリメンテーションがあります。
- * が付いた演算子は、TensorFlow のバージョンが 2.0 より新しいことを示します。

Caffe でサポートされる演算子

表 26: Caffe でサポートされる演算子

Caffe		XIR		DPU のインプリメンテーション
OP 名	属性	OP 名	属性	
input	shape	data	shape	入力データ用のメモリを割り当てます。
			data_type	
convolution	kernel_size:	conv2d (group = 1) / depthwise-conv2d (group = input channel)	kernel	group == input channel の場合、convolution は Depthwise たたみ込みエンジンにコンパイルされます。group == 1 の場合、convolution はたたみ込みエンジンにマップされます。それ以外の場合、CPU にマップされます。
	stride		stride	
	pad		pad	
			pad_mode (FLOOR)	
	dilation		dilation	
	bias_term			
	num_output			
	group			
deconvolution	kernel_size:	transposed-conv2d (group = 1) / depthwise-transposed-conv2d (group = input channel)	kernel	group == input channel の場合、deconvolution は Depthwise たたみ込みエンジンにコンパイルされます。group == 1 の場合、deconvolution はたたみ込みエンジンにマップされます。それ以外の場合、CPU にマップされます
	stride		stride	
	pad		pad	
			pad_mode (FLOOR)	
	dilation		dilation	
	bias_term			
	num_output			
	group			
innerproduct	bias_term	conv2d / matmul	transpose_a	inner-product は matmul に変換され、matmul は conv2d に変換されてたたみ込みエンジンにコンパイルされます。inner-product の変換がエラーになった場合は、CPU によってインプリメントされます。
	num_output		transpose_b	

表26: Caffeでサポートされる演算子(続き)

Caffe		XIR		DPUのインプリメンテーション
OP名	属性	OP名	属性	
scale	bias_term	depthwise-conv2d / scale		scaleはdepthwise-convolutionに変換されます。それ以外の場合、CPUにマップされます。
pooling	kernel_size:	maxpool2d (pool_method = 0) / avgpool2d (pool_method = 1)	kernel_size:	ブーリングエンジン
	stride		stride	
	global_pooling		global	
	pad		pad	
	pool_method		pad_mode(CEIL)	
			count_include_pad (true)	
			count_include_invalid (false)	
eltwise	coeff = 1	add		要素ごとの加算エンジン
	operation = SUM			
concat	axis	concat	axis	特殊な読み出しありまたは書き込みストラテジとオンチップメモリの慎重な割り当てにより、concatの結果によって生じるオーバーヘッドを低減します。
relu	negative_slope	relu / leakyrelu	alpha	アクティベーションは、convolution、addなどの隣接する動作に融合されます。
relu6		relu6		
fixneuron	bit_width	fix	bit_width	コンパイル中にfloat2fixとfix2floatに分割され、float2fixおよびfix2floatの動作は隣接する動作と融合されて粗粒度の動作に変換されます。
	quantize_pos		fix_point	
			if_signed	
			round_mode	
reshape	shape	reshape	shape	これらの動作はshapeに関連する動作であり、通常は削除されるか、またはreshapeに変換されます。これによるオンチップデータレイアウトへの影響はありません。それ以外の場合、CPUにコンパイルされます。
permute	order	reshape / transpose	order	
flatten	axis	reshape / flatten	start_axis	
	end_axis		end_axis	
reorg	strides	reorg	strides	reorgがハードウェア要件を満たす場合、reorgはDPUインプリメンテーションにマップされます。
	reverse		reverse	

表 26: Caffe でサポートされる演算子 (続き)

Caffe		XIR		DPU のインプリメンテーション
OP 名	属性	OP 名	属性	
deephiresize	scale	resize	size	resize のモードが「BILINEAR」、align_corner=false、half_pixel_centers = false、size = 2、4、8 である場合、align_corner=false、half_pixel_centers = true、size = 2、4 は、DPU インプリメンテーション (pad +depthwise-転置された conv2d) に変換できます。 resize のモードが「NEAREST」で size が整数である場合、resize は DPU インプリメンテーションにマップされます。
	mode		mode	
			align_corners=false	
			half_pixel_centers=false	
gstiling	strides	gstiling	stride	gstiling のストライドが整数である場合、gstiling は特殊な DPU 読み出し/書き込み命令にマップされることがあります。
	reverse		reverse	
slice	axis	strided_slice	begin	CPU インプリメンテーションにのみコンパイルされます。
	slice_point		end	
			strides	
priorbox	min_sizes	priorbox	min_sizes	
	max_sizes		max_sizes	
	aspect_ratio		aspect_ratio	
	flip		flip	
	clip		clip	
	variance		variance	
	step		step	
	offset		offset	
softmax	axis	softmax	axis	

PyTorch でサポートされる演算子

表 27: PyTorch でサポートされる演算子

PyTorch		XIR		DPU のインプリメンテーション
API	属性	OP 名	属性	
パラメーター	data	const	data	入力データ用のメモリを割り当てます。
			shape	
			data_type	

表27: PyTorchでサポートされる演算子(続き)

PyTorch		XIR		DPUのインプリメンテーション
API	属性	OP名	属性	
Conv2d	in_channels	conv2d (groups = 1) / depthwise-conv2d (groups = input channel)		groups == input channel の場合、 convolution は Depthwise たたみ込みエンジンにコンパイルされます。groups == 1 の場合、 convolution はたたみ込みエンジンにマップされます。それ以外の場合、CPU にマップされます。
	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)		groups == input channel の場合、 convolution は Depthwise たたみ込みエンジンにコンパイルされます。groups == 1 の場合、 convolution はたたみ込みエンジンにマップされます。それ以外の場合、CPU にマップされます。
	out_channels			
	kernel_size:		kernel	
	stride		stride	
	padding		pad	
	padding_mode('zeros')		pad_mode (FLOOR)	
	groups			
	dilation		dilation	
matmul		conv2d / matmul	transpose_a	matmul は conv2d に変換され、たたみ込みエンジンにコンパイルされます。matmul の変換がエラーになった場合は、CPU によってインプリメントされます。
			transpose_b	
MaxPool2d/ AdaptiveMaxPool2d	kernel_size:	maxpool2d	kernel	ブーリングエンジン
	stride		stride	
	padding		pad	
	ceil_mode		pad_mode	
	output_size (adaptive)		global	
AvgPool2d/ AdaptiveAvgPool2d	kernel_size:	avgpool2d	kernel	ブーリングエンジン
	stride		stride	
	padding		pad	
	ceil_mode		pad_mode	
	count_include_pad		count_include_pad	
			count_include_invalid(true)	
	output_size (adaptive)		global	

表 27: PyTorch でサポートされる演算子 (続き)

PyTorch		XIR		DPU のインプリメンテーション
API	属性	OP 名	属性	
ReLU		relu		アクティベーションは、 convolution、add などの隣接する動作に融合されます。
LeakyReLU	negative_slope	leakyrelu	alpha	
ReLU6		relu6		
Hardtanh	min_val = 0			
	max_val = 6			
ConstantPad2d/ ZeroPad2d	padding	pad	paddings	「CONSTANT」パディングは、隣接する動作と融合されます。
	value = 0		mode ("CONSTANT")	
add		add		add が要素ごとの加算である場合、add は DPU の要素ごとの加算エンジンにマップされます。add がチャネルごとの加算である場合、add と、convolution などの隣接する動作を融合できる可能性を探します。shape に関する動作である場合、コンパイル中に削除されます。粗粒度の動作のコンポーネントである場合、隣接する動作と融合されます。それ以外の場合、CPU インプリメンテーションにコンパイルされます。
sub / rsub		sub		
mul		mul		
max	dim	reduction_max	axis	
	keepdim		keep_dims	
mean	dim	reduction_mean	axis	resize のモードが「BILINEAR」、 align_corner=false、 half_pixel_centers = false、 size = 2、4、8 である場合、 align_corner=false、 half_pixel_centers = true、 size = 2、4 は、 DPU インプリメンテーション (pad +depthwise-転置された conv2d) に変換できます。 resize のモードが「NEAREST」で size が整数である場合、 resize は DPU インプリメンテーションにマップされます。
	keepdim		keep_dims	
interpolate / upsample / upsample_bilinear / upsample_nearest	size	resize	size	
	scale_factor			
	mode		mode	
	align_corners		align_corners	
			half_pixel_centers = ! align_corners	
transpose	dim0	transpose	order	reshape 動作に変換される場合があります。さらに、次元変換動作を隣接する動作の特殊な読み込み/保存命令に融合してオーバーヘッドを低減できる可能性を探します。それ以外の場合、CPU インプリメンテーションにマップされます。
	dim1			
permute	dims			
view	size	reshape	shape	
flatten	start_dim	reshape / flatten	start_axis	
	end_dim		end_axis	
squeeze	dim	reshape / squeeze	axis	

表27: PyTorchでサポートされる演算子(続き)

PyTorch		XIR		DPUのインプリメンテーション
API	属性	OP名	属性	
cat	dim	concat	axis	特殊な読み出しありは書き込みストラテジとオンチップメモリの慎重な割り当てにより、concatの結果生じるオーバーヘッドを低減します。
aten::slice*	dim	strided_slice		strided_sliceがshapeに関連する動作であるか、粗粒度の動作の構成要素である場合、strided_sliceは削除されます。それ以外の場合、strided_sliceはCPUインプリメンテーションにコンパイルされます。
	start		begin	
	end		end	
	step		strides	
BatchNorm2d	eps	depthwise-conv2d / batchnorm	epsilon	batch_normが量子化され、等価なdepthwise-conv2dに変換可能である場合、batch_normはdepthwise-conv2dに変換されます。コンパイラは、batch_normをDPUインプリメンテーションにマップするためのコンパイルができる可能性を探します。それ以外の場合、batch_normはCPUによって実行されます。
			axis	
			moving_mean	
			moving_var	
			gamma	
			beta	
softmax	dim	softmax	axis	CPUインプリメンテーションにのみコンパイルされます。
Tanh		tanh		
Sigmoid		sigmoid		

注記:

- PyTorchのテンソルのスライスがPython構文で書かれている場合、`aten::slice`に変換されます。

DPUCADX8Gによるコンパイル

このセクションでは、DPUCADX8G(以前はxfDNN)フロントエンドコンパイラについて簡単に説明します。ここでは、共通の中間表現を用いて構築したCaffeおよびTensorFlowインターフェイスを紹介します。これらのインターフェイスはすべてのDPUで共通です。

このセクションでは、サンプル(ソフトウェアディストリビューション)、モデルの量子化、および進行中のサブグラフと組み合わせて使用する実行手順についても説明します。コンパイラはオープンソースとして提供されているため、ここからさらに情報が得られます。

この新しい環境に慣れることを目的としているため、必要な手順と一部の内容のみを提示します。設計環境が設定されていることを前提とし、分類ネットワークなどを検討している方を対象にDPUCADX8Gデザインで実行するための生成手順を説明します。

最終的な目標が FPGA コードの検査と時間の見積もりである場合は、コンパイラを単独で使用できます。最終的な目標が FPGA インスタンスでネットワークを実行することである場合は、DPUCADX8G コンパイラをパーティショナーと組み合わせて使用する必要があります。次の章では、これを目的とした 2 つのツールを紹介します。1 つは Caffe 用で、もう 1 つは TensorFlow 用です。Caffe の場合、パーティショナーはコンパイラ出力を直接使用してランタイムに提供できます。これは、パーティショナーが単一の FPGA サブグラフ内で計算を分割しているからです。TensorFlow パーティショナーは複数のサブグラフを使用します。

Caffe

プレゼンテーションのために、MODEL (model.prototxt)、WEIGHT (model.caffemodel)、および QUANT_INFO (量子化情報ファイル) があることを前提とします。基本の Caffe コンパイラインターフェイスには、シンプルなヘルプがあります。

```
vai_c_caffe -help
*****
* VITIS-AI Compilation - Xilinx Inc.
*****
usage: vai_c_caffe.py [-h] [-p PROTOTXT] [-c CAFFEMODEL] [-a ARCH]
                      [-o OUTPUT_DIR] [-n NET_NAME] [-e OPTIONS]optional
arguments:
  -h, --help            show this help message and exit
  -p PROTOTXT, --prototxt PROTOTXT
                        prototxt
  -c CAFFEMODEL, --caffemodel CAFFEMODEL
                        caffe-model
  -a ARCH, --arch ARCH json file
  -o OUTPUT_DIR, --output_dir OUTPUT_DIR
                        output directory
  -n NET_NAME, --net_name NET_NAME
                        prefix-name for the outputs
  -e OPTIONS, --options OPTIONS
                        extra options
```

このインターフェイスの主な目標は、異なるデザインで最小限の措置を指定することです。ここでは、最小限の入力でスタートする、DPUCADX8G デザイン用の実行方法を説明します。

```
vai_c_caffe.py -p MODEL -c WEIGHT -a vai/DPUCADX8G/tools/compile/arch.json -
-o WORK -n cmd -e OPTIONS
```

MODEL、WEIGHT、出力の書き込み先、生成されるコードの名前(すなわち、cmd)を指定します。これにより、WORK ディレクトリに 4 つの出力ファイルが作成されます。

```
compiler.json  quantizer.json  weights.h5  meta.json
```

これがランタイムとの主なコントラクトです。JSON ファイルは 3 つあります。実行する手順に関する情報が含まれているファイルと、量子化の情報(スケール/シフトの方法など)が含まれているファイルのほかに、arch.json ファイルから作成される meta.json ファイルがあります。これは、基本的にはランタイム情報を指定する辞書ファイルです。名前 cmd は必要ですが、Vitis AI ランタイムでは使用されません。

DPU のほかのバージョンとの主な違いは、オプションを使用して QUANT_INFO を指定する必要があることです。

```
-e "{'quant_cfgfile' : '/SOMEWHERE/quantize_info.txt'}"
```

オプション フィールドは、Python の辞書を表す文字列です。この例では、[第4章: モデルの量子化](#) で説明した、個別に計算された量子化ファイルの場所を指定します。コンテキストでは、ほかの DPU バージョンはこの情報をモデルまたは重みのいずれかで構築します。したがって、拡張モデルはバニラ Caffe モデルではないため、これらを実行するにはカスタム Caffe が必要です。DPUCADX8G は、ネイティブ Caffe (およびカスタム Caffe) を使用して実行します。

注記: 量子化ファイルを提供する必要があることに注意してください。コンパイラは量子化ファイルを要求します。このときにコンパイラが探すことになると最終的にクラッシュします。完全な Caffe モデルには、prototxt と caffemodel の両方が必要です。arch.json ファイルについては後ほど説明しますが、これも必要です。これはスクリプト形式を使用する統合 Caffe インターフェイスですが、より高度なカスタマイズやコンパイルを可能にする Python インターフェイスがあるため、アドバンスユーザーはモデルのさらなる最適化が可能になります。

TensorFlow

Caffe と TensorFlow の主な違いは、モデルが単一ファイルで定義され、量子化情報を GraphDef から取得する必要があるという点です。

```
*****
* VITIS_AI Compilation - Xilinx Inc.
*****
usage: vai_c_tensorflow.py [-h] [-f FROZEN_PB] [-a ARCH] [-o OUTPUT_DIR]
                           [-n NET_NAME] [-e OPTIONS] [-q]

optional arguments:
  -h, --help            show this help message and exit
  -f FROZEN_PB, --frozen_pb FROZEN_PB
                        prototxt
  -a ARCH, --arch ARCH json file
  -o OUTPUT_DIR, --output_dir OUTPUT_DIR
                        output directory
  -n NET_NAME, --net_name NET_NAME
                        prefix-name for the outputs
  -e OPTIONS, --options OPTIONS
                        extra options
  -q, --quant_info     extract quant info
```

ここで、インターフェイスは凍結されたグラフの指定方法を明確に定義しています。モデルと量子化情報が必要であると想定します。

```
vai_c_tensorflow.py --frozen_pb deploy.pb --net_name cmd --options
  "{'placeholdershape': {'input_tensor' : [1,224,224,3]}, 'quant_cfgfile':
  'fix_info.txt'}" --arch arch.json --output_dir work/temp
```

ここに示されているとおり、量子化情報と入力プレースホルダーの形状が指定されています。通常、モデルの入力を指定するプレースホルダー レイヤーを使用します。すべてのサイズを指定し、バッチ数 (= 1) を使用することを推奨しています。レイテンシを最適化し、バッチ サイズ 1 ~ 4 を受け入れます (ただし、これによりレイテンシは改善されず、スループットもほとんど改善されず、どのネットワークに対しても完全なテストは実施されない)。

キャリブレーションと微調整では、ネイティブ TensorFlow で実行できないモデルが提供される場合がありますが、量子化情報が含まれています。[-q, --quant_info extract quant info] を使用してフロントエンドを実行すると、量子化情報が作成されます。

ソフトウェアリポジトリには、コンパイラが 2 回呼び出される例が提供されています。1 回目の呼び出しで量子化情報ファイルを作成し (デフォルトの名前と場所を使用)、これをコード生成の入力として使用します。

注記: 出力ディレクトリと生成されるコード名を必ず提供してください。ランタイム コントラクトは、出力が書き込まれる場所に基づいています。アーキテクチャごとに異なるコンパイラを呼び出す主な方法として、arch.json ファイルの使用があります。このファイルは、出力記述用のテンプレートとして、およびプラットフォーム/ターゲット FPGA デザインの内部機能として使用されます。さらに、アドバンス ユーザーが独自の最適化を利用できる Python インターフェイスもあります。

VAI_C の使用法

Caffe および TensorFlow フレームワークに対応する Vitis AI コンパイラは、それぞれ vai_c_caffe と vai_c_tensorflow であり、クラウドおよびエッジ DPU で利用可能です。VAI_C の一般的なオプションを次の表に示します。

表 28: クラウドおよびエッジ DPU 用の VAI_C の一般的なオプション

パラメーター	説明
--arch	JSON 形式の VAI_C コンパイラの DPU アーキテクチャ コンフィギュレーション ファイル。コンパイル時のクラウドおよびエッジ DPU 専用のオプションが含まれています。
--prototxt	コンパイラ vai_c_caffe 用の Caffe prototxt ファイルのパス。このオプションは、vai_q_caffe で生成された量子化された Caffe モデルをコンパイルするときにのみ必要です。
--caffemodel	コンパイラ vai_c_caffe 用の Caffe caffemodel ファイルのパス。このオプションは、vai_q_caffe で生成された量子化された Caffe モデルをコンパイルするときにのみ必要です。
--frozen_pb	コンパイラ vai_c_tensorflow 用の TensorFlow の凍結された protobuf ファイルのパス。このオプションは、vai_q_tensorflow で生成された量子化された TensorFlow モデルでのみ必要です。
--output_dir	コンパイル プロセス後の vai_c_caffe および vai_c_tensorflow の出力ディレクトリのパス。
--net_name	VAI_C でコンパイルされた後のネットワーク モデルの DPU カーネルの名前。
--options	'key': 'value' 形式で、クラウドまたはエッジ DPU 用の追加オプションのリスト。指定するオプションが複数ある場合は、「','」で区切られます。追加のオプションに値がない場合は、空の文字列を指定する必要があります。例: --options "{'cpu_arch':'arm32', 'dcf':'/home/edge-dpu/zynq7020.dcf', 'save_kernel':''}" 注記: 「--options」で指定された引数は、優先度が最も高いため、ほかの場所で指定された値を書き換えます。たとえば、「--options」で dcf が指定されている場合、JSON ファイルで指定されている値が書き換えられます。

モデルの運用と実行

Alveo U200/250 上でのモデルの運用と実行

Vitis AI は、FPGA 上でモデルを運用するための、エッジおよびクラウド用の統合された C++/Python API を提供します。

C++ API の詳細は、<https://github.com/Xilinx/Vitis-AI/blob/master/docs/DPUCADX8G/Vitis-C%2B%2BAPI.md> を参照してください。

Python API の詳細は、<https://github.com/Xilinx/Vitis-AI/blob/master/docs/DPUCADX8G/Vitis-PythonAPI.md> を参照してください。

VART を使用したプログラミング

Vitis AI は、次のインターフェイスを備える C++ DpuRunner クラスを提供します。

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

注記: 歴史的な理由から、この関数は、実際には非同期ノンブロッキング関数ではなくブロッキング関数です。

1. 実行用の入力テンソルと結果を保存するための出力テンソルを作成します。ホスト ポインターは、TensorBuffer オブジェクトを使用して渡されます。この関数は、ジョブ ID と関数呼び出しのステータスを返します。

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

execute_async によって返されたジョブ ID は wait() に渡され、ジョブが完了して結果が準備できるまでブロックします。

```
TensorFormat get_tensor_format()
```

2. 必要なテンソル形式について DpuRunner に問い合わせます。

DpuRunner::TensorFormat::NCHW または DpuRunner::TensorFormat::NHWC を返します。

```
std::vector<Tensor*> get_input_tensors()
```

3. 読み込まれた Vitis AI モデルに必要な出力テンソルの形状と名前について DpuRunner に問い合わせます。

```
std::vector<Tensor*> get_output_tensors()
```

4. DpuRunner オブジェクトを作成するため、次を呼び出します。

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

次の結果が返されます。

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

create_runner への入力は、Vitis AI コンパイラで生成された XIR サブグラフです。



ヒント: VART でマルチスレッド処理を有効にするには、スレッドごとにランナーを作成します。

C++ の例

```
// 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 は、C DpuRunner のインプリメンテーションを使用して、C++ クラスに類似する Python ctypes Runner クラスも提供しています。

```
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 の例

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

VARTによるDPUのデバッグ

このセクションの目的は、VARTツールを使用してDPUの推論結果を検証する方法を示すことです。例として、TensorFlow ResNet50、Caffe ResNet50、およびPyTorch ResNet50の各ネットワークを使用します。VARTを使用してDPUをデバッグするための4つの手順を次に示します。

1. 量子化された推論モデルとリファレンス結果を生成する
2. DPU xmodelを生成する
3. DPUの推論結果を生成する
4. リファレンス結果とDPUの推論結果をクロスチェックする

DPUの結果のデバッグを始める前に、[第2章: クイックスタート](#)の手順に従って環境が設定されていることを確認してください。

TensorFlowのワークフロー

量子化された推論モデルとリファレンス結果を生成するには、次の手順に従います。

1. 次のコマンドを実行してモデルを量子化することにより、量子化された推論モデルを生成します。

量子化されたモデル(`quantize_eval_model.pb`)は、`quantize_model`フォルダーに生成されます。

```
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. 次のコマンドを実行してリファレンスデータを生成することにより、リファレンス結果を生成します。

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

次の図に、リファレンスデータの一部を示します。

```
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. 次のコマンドを実行して DPU の xmodel ファイルを生成することにより、DPU の xmodel を生成します。

```
vai_c_tensorflow --frozen_pb quantize_model/quantize_eval_model.pb \
--arch /opt/vitis_ai/compiler/arch/DPUCAHX8H/U50/arch.json \
--output_dir compile_model \
--net_name resnet50_tf
```

4. 次のコマンドを実行して DPU の推論結果を生成することにより、DPU の推論結果を生成し、DPU の推論結果とリファレンスデータを自動的に比較します。

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

xilinx_test_dpu_runner の使用方法は次のとおりです。

```
xilinx_test_dpu_runner <model_file> <input_data>
```

上記のコマンドの実行後、DPU の推論結果および比較結果 result.log が生成されます。DPU の推論結果は、dump フォルダーに置かれます。

5. リファレンス結果と DPU の推論結果をクロスチェックします。

- a. すべてのレイヤーについて 比較結果を表示します。

```
grep --color=always 'XLNX_GOLDEN_DIR.*layer_name' result.log
```

```

I1019 02:21:32.894465 39399 dpu_runner_base_imp.cpp:458} XLNX_GOLDEN_DIR: compare data success !layer_name resnet_v1_50_block4_unit_2_bottleneck_v1_conv2_Relu_aquant dump_md5 3a5
ffeaa9fe9f3d485a22e632175c5a527
I1019 02:21:32.895944 dpu_runner_base_imp.cpp:458} XLNX_GOLDEN_DIR: compare data success !layer_name resnet_v1_50_block4_unit_2_bottleneck_v1_Relu_aquant dump_md5 caff752e3
9c6caad5faaa0426d69379cb
I1019 02:21:32.912925 39399 dpu_runner_base_imp.cpp:458} XLNX_GOLDEN_DIR: compare data success !layer_name resnet_v1_50_block4_unit_2_bottleneck_v1_Relu_aquant dump_md5 caff752e3
I1019 02:21:32.923344 39399 dpu_runner_base_imp.cpp:458} XLNX_GOLDEN_DIR: compare data success !layer_name resnet_v1_50_block4_unit_3_bottleneck_v1_conv1_Relu_aquant dump_md5 0ba
234559c87a36094a87d1546683b8
I1019 02:21:32.932585 39399 dpu_runner_base_imp.cpp:458} XLNX_GOLDEN_DIR: compare data success !layer_name resnet_v1_50_block4_unit_3_bottleneck_v1_conv1_Relu_aquant dump_md5 0ba
234559c87a36094a87d1546683b8
I1019 02:21:32.943107 39399 dpu_runner_base_imp.cpp:458} XLNX_GOLDEN_DIR: compare data success !layer_name resnet_v1_50_block4_unit_3_bottleneck_v1_conv2_Relu_aquant dump_md5 61f
98a356656d7d7cfbc8d36822268f
I1019 02:21:32.952344 39399 dpu_runner_base_imp.cpp:458} XLNX_GOLDEN_DIR: compare data success !layer_name resnet_v1_50_block4_unit_3_bottleneck_v1_Relu_aquant dump_md5 caff752e3
9c6caad5faaa0426d69379cb
I1019 02:21:32.964639 39399 dpu_runner_base_imp.cpp:458} XLNX_GOLDEN_DIR: compare data success !layer_name resnet_v1_50_block4_unit_3_bottleneck_v1_conv2_Relu_aquant dump_md5 61f
98a356656d7d7cfbc8d36822268f
I1019 02:21:32.980353 39399 dpu_runner_base_imp.cpp:458} XLNX_GOLDEN_DIR: compare data success !layer_name resnet_v1_50_block4_unit_3_bottleneck_v1_Relu_aquant dump_md5 46e918e56
611ca0d6c626bcf563a7c1
I1019 02:21:32.993969 39399 dpu_runner_base_imp.cpp:458} XLNX_GOLDEN_DIR: compare data success !layer_name resnet_v1_50_pool5_mul_aquant dump_md5 704b0f6f810a788ba6b71b3b846cfb85
f11ca0d6c626bcf563a7c1
I1019 02:21:33.001917 39399 dpu_runner_base_imp.cpp:458} XLNX_GOLDEN_DIR: compare data success !layer_name resnet_v1_50_pool5_mul_aquant dump_md5 704b0f6f810a788ba6b71b3b846cfb85
I1019 02:21:33.017825 39399 dpu_runner_base_imp.cpp:458} XLNX_GOLDEN_DIR: compare data success !layer_name resnet_v1_50_logs_BiasAdd_aquant dump_md5 68f0e3830b2084a36c84adc5b8e
183e8

```

- b. エラーのあるレイヤーのみ表示します。

```
grep --color=always 'XLNX_GOLDEN_DIR.*fail ! layer_name' result.log
```

クロスチェックがエラーになった場合は、次の方法でどのレイヤーが原因でエラーになったかをさらに確認します。

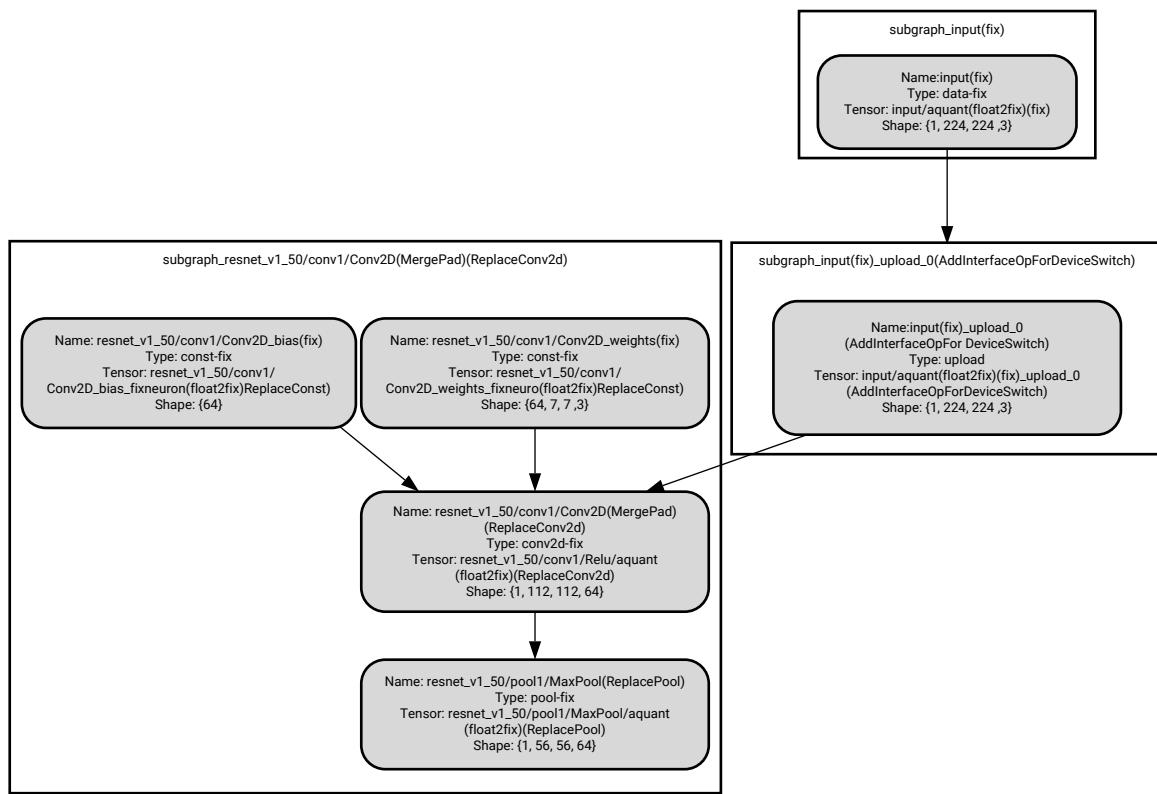
- DPU と GPU の入力をチェックし、同じ入力データを使用していることを確認します。
- xir ツールを使用して、ネットワークの構造を表示する画像を生成します。

```
Usage: xir svg <xmodel> <svg>
```

注記: Vitis AI Docker 環境で、次のコマンドを実行して必要なライブラリをインストールします。

```
sudo apt-get install graphviz
```

作成した画像を開くと、これらの op の周囲に複数の小さいボックスが表示されています。各ボックスは DPU 上のレイヤーを意味します。最後の op の名前を使用して、GPU のダンプ結果からそれに対応する op を検索できます。次の図に、構造の一部を示します。



X24898-120920

c. ファイルをザイリンクスに送信します。

DPU 上で特定のレイヤーに問題があるとわかった場合は、量子化されたモデル (quantize_eval_model.pb など) を、ザイリンクス側でさらに解析するための 1 つのパッケージとして準備し、詳細な説明と共にザイリンクスに送信します。

Caffe ワークフロー

量子化された推論モデルとリファレンス結果を生成するには、次の手順に従います。

1. 次のコマンドを実行してモデルを量子化することにより、量子化された推論モデルを生成します。

```
vai_q_caffe quantize -model float/test_quantize.prototxt \
-weights float/trainval.caffemodel \
-output_dir quantize_model \
-keep_fixed_neuron \
2>&1 | tee ./log/quantize.log
```

次のファイルが quantize_model フォルダーに生成されます。

- deploy.caffemodel
- deploy.prototxt
- quantize_train_test.caffemodel
- quantize_train_test.prototxt

2. 次のコマンドを実行してリファレンス データを生成することにより、リファレンス結果を生成します。

```
DECENT_DEBUG=5 vai_q_caffe test -model quantize_model/dump.prototxt \
 -weights quantize_model/quantize_train_test.caffemodel \
 -test_iter 1 \
 2>&1 | tee ./log/dump.log
```

これにより、次の図に示すように、dump_gpu フォルダーおよびファイルが作成されます。

bn2a_branch1.bin	bn5b_branch2b.bin	res2c_branch2a_relu.bin
bn2a_branch2a.bin	bn5b_branch2c.bin	res2c_branch2a_weights.bin
bn2a_branch2b.bin	bn5c_branch2a.bin	res2c_branch2b.bin
bn2a_branch2c.bin	bn5c_branch2b.bin	res2c_branch2b_bias.bin
bn2b_branch2a.bin	bn5c_branch2c.bin	res2c_branch2b_relu.bin
bn2b_branch2b.bin	bn_conv1.bin	res2c_branch2b_weights.bin
bn2b_branch2c.bin	conv1.bin	res2c_branch2c.bin
bn2c_branch2a.bin	conv1_bias.bin	res2c_branch2c_bias.bin
bn2c_branch2b.bin	conv1_relu.bin	res2c_branch2c_weights.bin
bn2c_branch2c.bin	conv1_weights.bin	res2c_relu.bin
bn3a_branch1.bin	data.bin	res3a.bin
bn3a_branch2a.bin	fc1000.bin	res3a_branch1.bin
bn3a_branch2b.bin	pool1.bin	res3a_branch1_bias.bin
bn3a_branch2c.bin	pool5.bin	res3a_branch1_weights.bin
bn3b_branch2a.bin	prob.bin	res3a_branch2a.bin
bn3b_branch2b.bin	res2a.bin	res3a_branch2a_bias.bin
bn3b_branch2c.bin	res2a_branch1.bin	res3a_branch2a_relu.bin
bn3c_branch2a.bin	res2a_branch1_bias.bin	res3a_branch2a_weights.bin

3. 次のコマンドを実行して DPU の xmodel ファイルを生成することにより、DPU の xmodel を生成します。

```
vai_c_caffe --prototxt quantize_model/deploy.prototxt \
 --caffemodel quantize_model/deploy.caffemodel \
 --arch /opt/vitis_ai/compiler/arch/DPUCAHX8H/U50/arch.json \
 --output_dir compile_model \
 --net_name resnet50
```

4. 次のコマンドを実行して DPU の推論結果を生成します。

```
env XLNX_ENABLE_DUMP=1 XLNX_ENABLE_DEBUG_MODE=1 \
 xilinx_test_dpu_runner ./compile_model/resnet50.xmodel \
 ./dump_gpu/data.bin 2>result.log 1>&2
```

xilinx_test_dpu_runner の使用方法は次のとおりです。

```
xilinx_test_dpu_runner <model_file> <input_data>
```

上記のコマンドの実行後、DPU の推論結果および比較結果 result.log が生成されます。DPU の推論結果は、dump フォルダーに置かれます。

5. リファレンス結果と DPU 推論結果をクロスチェックします。

クロスチェック メカニズムは、1つのレイヤーへの入力がリファレンスと同一であることを確認してから、出力もリファレンスと同一であることを確認します。これは diff、vimdiff、および cmp などのコマンドを使用して実行されます。2つのファイルが同一である場合、diff と cmp は、コマンド ラインに何も返しません。

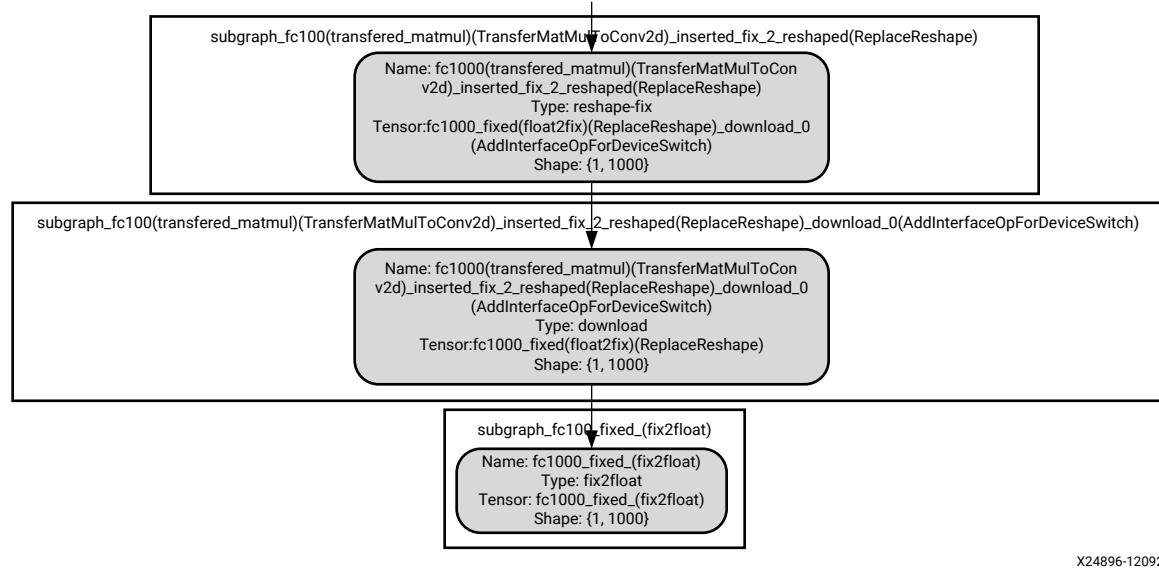
- DPU と GPU の入力をチェックし、同じ入力データを使用していることを確認します。
- xir ツールを使用して、ネットワークの構造を表示する画像を生成します。

```
Usage: xir svg <xmodel> <svg>
```

注記: Vitis AI Docker 環境で、次のコマンドを実行して必要なライブラリをインストールします。

```
sudo apt-get install graphviz
```

次の図に、xir_cat によって生成される ResNet50 モデル構造の一部を示します。



- c. xmodel 構造のイメージを表示し、モデルの最後のレイヤー名を見つけます。



推奨: 最後のレイヤーを最初にチェックします。最後のレイヤーのクロスチェックが正常に終了した場合は、レイヤー全体のクロスチェックが合格になるため、その他のレイヤーのクロスチェックは不要です。

このモデルの場合、最後のレイヤーの名前は「subgraph_fc1000_fixed_(fix2float)」です。

- dump_gpu および dump フォルダーの下でキーワード fc1000 を検索します。dump_gpu フォルダーの下にリファレンス結果ファイル fc1000.bin が、dump/subgraph_fc1000/output/ フォルダーの下に DPU 推論結果 0.fc1000_inserted_fix_2.bin が確認できます。
 - 2つのファイルに対して diff コマンドを実行します。
最後のレイヤーのクロスチェックがエラーになった場合は、クロスチェックを最初のレイヤーから実行して、クロスチェックがエラーになるレイヤーが確認できるまでチェックを続ける必要があります。
- 注記:** レイヤーの入力または出力が複数ある場合は(例: res2a_branch1)、入力に問題がないかチェックしてから、出力に問題がないかチェックします。
- DPU クロスチェックがエラーになる場合は、ザイリンクスにファイルを送信してください。
DPU 上で特定のレイヤーに問題があるとわかった場合は、ザイリンクスでさらに解析するために、次のファイルを1つのパッケージとして準備し、詳細な説明と共にザイリンクスに送信します。
 - 浮動小数点モデルおよび prototxt ファイル
 - 量子化されたモデル(deploy.caffemodel、deploy.prototxt、quantize_train-test.caffemodel、および quantize_train-test.prototxt など)。

PyTorch ワークフロー

量子化された推論モデルとリファレンス結果を生成するには、次の手順に従います。

- 次のコマンドを実行してモデルを量子化することにより、量子化された推論モデルを生成します。

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

2. 次のコマンドを実行してリファレンス データを生成することにより、リファレンス結果を生成します。

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

3. 次のコマンドを実行して DPU の xmodel ファイルを生成することにより、DPU の xmodel を生成します。

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

4. DPU の推論結果を生成します。

この手順は、Caffe ワークフローの手順と同じです。

5. リファレンス結果と DPU 推論結果をクロスチェックします。

この手順は、Caffe ワークフローの手順と同じです。

複数 FPGA のプログラミング

多くの最新型サーバーには複数のザイリンクス Alveo™ カードが搭載されており、深層学習の推論性能を拡大/向上させることを目的としています。Vitis AI は、次の構築ブロックを使用して、複数 FPGA サーバーをサポートします。

Xbutler

Xbutler ツールは、マシン上のザイリンクス FPGA リソースを管理および制御します。Vitis AI 1.0 リリースでは、Xbutler を使用して深層学習ソリューションを実行するには、Xbutler のインストールが必須です。Xbutler は、サーバーとクライアントのパラダイムとして実装されます。Xbutler は、ザイリンクス XRT のアドオン ライブラリであり、複数 FPGA のリソース管理を容易にします。Xbutler は、ザイリンクス XRT の代替製品ではありません。Xbutler の機能一覧は次のとおりです。

- 複数 FPGA のヘテロジニアス サポートを有効化
- C++/Python API および CLI (クライアントによるリソースの割り当て、使用、解放に使用)
- FPGA でのリソース割り当て、計算ユニット (CU)、およびサービス粒度の有効化
- リソースの自動解放
- マルチクライアント サポート: マルチクライアント/ユーザー/プロセスのリクエストを有効化
- XCLBIN から DSA への自動関連付け
- クライアント/ユーザー間のリソース共有
- コンテナー化されたサポート
- ユーザー定義関数
- ロギング サポート

Vitis AI での複数の FPGA、複数のグラフの運用

Vitis AI は、単一/複数の FPGA で複数のモデルを運用するための統合されたランナー API を使用して構築された各種のアプリケーションを提供します。詳しい説明および例は、Vitis AI GitHub ([複数のテナント、複数の FPGA の運用](#)) にあります。

Xstream API

典型的なエンドツーエンドのワークフローには、ML、ビデオ、データベースのアクセラレーションなどの高速化サービス用の FPGA と、FPGA に実装されていない外部回路用の I/O やコンピューティング用 CPU で構成されるヘテロジニアス コンピューティング ノードが含まれます。Vitis AI は、ストリーミング アプリケーションを Python で作成できるようにする API と関数のセットを提供します。Xstream API は、Xbutler で提供される機能の上に構築されます。Xstream API のコンポーネントは次のとおりです。

- Xstream: Xstream (`$VAI PYTHON DIR/vai/dpuv1/rt/xstream.py`) は、複数プロセス間のデータストリーミングや実行フローと依存関係を制御するための一般的なメカニズムを提供します。
- Xstream チャネル: チャネルは英数字の文字列で定義されます。Xstream ノードは、チャネルにペイロードを公開し、チャネルをサブスクライブしてペイロードを受信できます。デフォルトのパターンは PUB-SUB です。つまり、チャネルのすべてのサブスクリバーは、そのチャネルに公開されたすべてのペイロードを受信します。これらのペイロードは、サブスクリバーがキューをすべて処理するまで、FIFO 順にサブスクリバーのキューに入れられます。
- Xstream ペイロード: ペイロードには、バイナリデータ(バイナリプロップ)とメタデータという2つの情報が含まれます。バイナリプロップとメタデータは、オブジェクトストアとして Redis を使用して送信されます。バイナリプロップは、大規模データを対象としています。メタデータは、ID、引数、オプションなどの小さなデータ用です。オブジェクト ID は ZMQ を介して送信されます。ZMQ は、ストリームフロー制御に使用されます。メタデータには ID フィールドが必要です。送信の終了を通知するために空のペイロードが使用されます。
- Xstream ノード: 各 Xstream ノードはストリームプロセッサとなります。ゼロから複数の入力チャネルをサブスクライブでき、またゼロから複数の出力チャネルに出力できる個別プロセスです。ノードは、その入力チャネルで受信したペイロードの計算を実行できます。計算は、CPU、FPGA、または GPU で実行できます。新しいノードを定義するには、`vai/dpuv1/rt/xsnodes` に新しい Python ファイルを追加してください。`ping.py` に例を示します。すべてのノードは、構築時に永久にループする必要があります。ループの各反復で、入力チャネルからペイロードを受信し、出力チャネルにペイロードを送信する必要があります。空のペイロードを受信した場合、ノードは `xstream.end()` を呼び出して空のペイロードを出力チャネルに転送してから終了する必要があります。
- Xstream グラフ: `$VAI PYTHON DIR/vai/dpuv1/rt/xsnodes/grapher.py` を使用して、1つまたは複数のノードを含むグラフを作成します。`Graph.serve()` が呼び出されると、グラフは各ノードを個別のプロセスとして生成し、それらの入出力チャネルを接続します。グラフはすべてのノードを管理します。グラフの例は、`neptune/services/ping.py` を参照してください。次に例を示します。

```
graph = grapher.Graph("my_graph")
graph.node("prep", pre.ImagenetPreProcess, args)
graph.node("fpga", fpga.FpgaProcess, args)
graph.node("post", post.ImagenetPostProcess, args)

graph.edge("START", None, "prep")
graph.edge("fpga", "prep", "fpga")
graph.edge("post", "fpga", "post")
graph.edge("DONE", "post", None)

graph.serve(background=True)
...
graph.stop()
```

- Xstream ランナー: グラフの入力チャネルにペイロードをプッシュする便利なクラスです。ペイロードは固有の ID を使用して送信されます。その後、ランナーは、送信された ID と一致するグラフの出力ペイロードを待ちます。このランナーは、ブロッキング関数呼び出しのルックアンドフィールを提供することが目的です。Xstream の完全スタンドアロンの例は、 `${VAI_ALVEO_ROOT}/examples/deployment_modes/xs_classify.py` です。

AI カーネル スケジューラ

実際の深層学習アプリケーションでは、多段データ処理パイプラインを使用して、計算量の多い各種の前処理(ディスクからのデータの読み込み、デコード、サイズ変更、色空間変換、スケーリングなど)、各種のMLネットワーク(CNNなど)、および各種の後処理(NMSなど)を実行する必要があります。

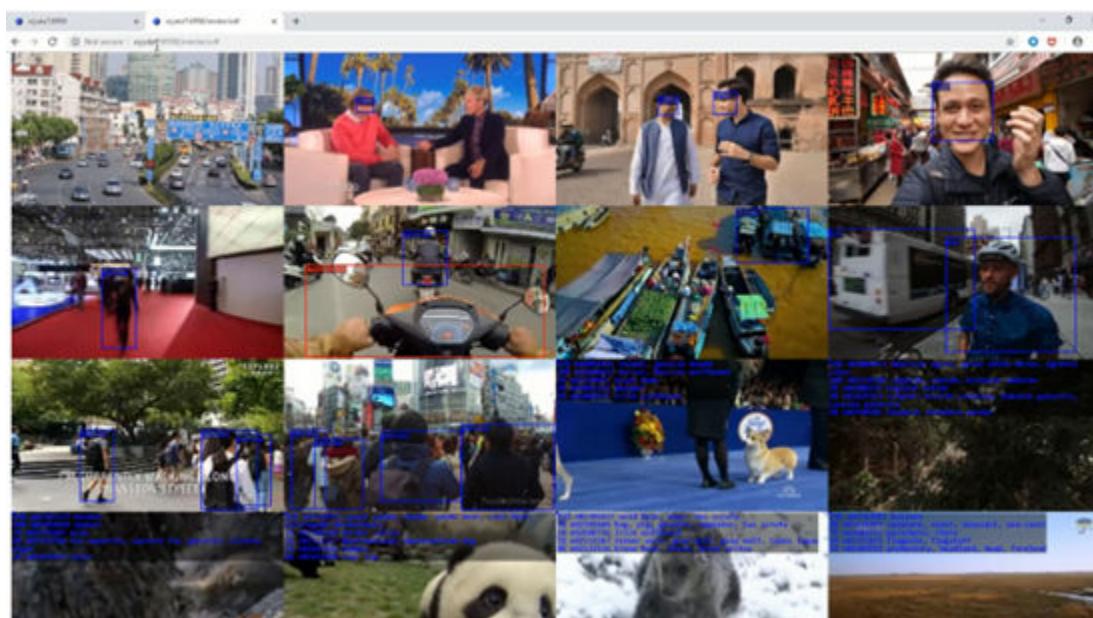
AI カーネル スケジューラ(AKS)は、前述のグラフを自動的かつ効率的にパイプライン処理するアプリケーションであり、ユーザーの労力は大幅に軽減されます。このスケジューラは、複雑なグラフの各段階に対応する各種のカーネルを提供します。これらのカーネルは、プラグアンドプレイ方式で柔軟な設定が可能です。たとえば、画像のデコードおよびサイズ変更などの前処理カーネル、Vitis AI の DPU カーネルなどの CNN カーネル、SoftMax や NMS などの後処理カーネルがあります。カーネルを使用してグラフを作成し、ジョブをスムーズに実行して最大限の性能が得られます。

詳細と例については、Vitis AI の GitHub ([AI カーネル スケジューラ](#)) を参照してください。

Neptune

Neptune は、Python で定義されたモジュール型ノードをウェブサーバーに提供します。これらのノードをグラフでつなぎ合わせることで、サービスを構築できます。これらのサービスを開始/停止するには、サーバーと通信します。独自のノードやサービスを追加すると、Neptune を拡張できます。Neptune は、Xstream API の上に構築されます。次の図では、ユーザーが YouTube の 16 本のビデオ上で 3 つの異なる機械学習モデルをリアルタイムで実行しています。単一の Neptune サーバーを介して、FPGA リソースの時間と空間を多重化します。詳細な資料およびサンプルは、\${VAI_ALVEO_ROOT}/neptune から入手できます。Neptune は、この Vitis AI リリースではアーリー アクセス段階です。

図 28: Alveo を利用したマルチストリームおよびマルチネットワーク プロセッシング



詳細は、[Vitis AI GitHub \(Neptune\)](#) を参照してください。

Apache TVM および Microsoft ONNX ランタイム

VART および関連 API 以外に、Vitis AI は Apache TVM および Microsoft ONNX Runtime フレームワークとも統合されており、モデルのサポートおよび自動パーティショニング機能が向上しています。この統合により、標準の Vitis AI コンパイラおよびクオンタイザーでは利用できない、コミュニティベースの機械学習フレームワーク インターフェイスが組み込まれています。また、特定のレイヤーがザイリンクスの DPU 上で利用できない場合のために、x86 および Arm CPU 向けの高度に最適化された CPU コードが組み込まれています。

TVM は、現在次の DPU でサポートされています。

- DPUCADX8G
- DPUCZDX8G

ONNX Runtime は、現在次の DPU でサポートされています。

- DPUCADX8G

Apache TVM

Apache TVM は、広範囲にわたるハードウェア アーキテクチャの効率的なインプリメンテーションの構築に焦点を合わせた、オープンソースの深層学習コンパイラ スタックです。これには、TensorFlow、TensorFlow Lite (TFLite)、Keras、PyTorch、MxNet、ONNX、Darknet やその他のフレームワークからのモデル解析が含まれます。TVM との Vitis AI の統合により、Vitis AI はこれらのフレームワークから得られるモデルを実行できます。TVM は 2 つのフェーズで構成されます。最初のフェーズはモデルのコンパイル/量子化で、ターゲット CPU および DPU 用の CPU/FPGA バイナリを生成します。2 番目のフェーズでは、TVM ランタイムをクラウドまたはエッジ デバイスにインストールし、Python または C++ の TVM API を呼び出してモデルを実行します。

Apache TVM の詳細は、<https://tvm.apache.org> を参照してください。

Vitis AI と TVM の統合に関する Vitis AI のチュートリアルとインストール ガイドは、Vitis AI の GitHub リポジトリ (<https://github.com/Xilinx/Vitis-AI/tree/master/external/tvm>) にあります。

Microsoft ONNX Runtime

Microsoft ONNX Runtime は、ONNX モデル向けのオープンソースの推論アクセラレータです。このプラットフォームは Vitis AI に統合され、さまざまなトレーニング フレームワークから移植可能な ONNX モデルを高度にサポートします。非常に使いやすい Python および C++ のランタイム API が組み込まれ、TVM に必要なコンパイル フェーズを別途実行しなくてもモデルをサポートできます。ONNXRuntime には、CPU と FPGA 間のパーティショニングを自動的に実行するパーティショナーが含まれ、モデルの運用がさらに容易になります。また、Vitis AI クオンタイザーも組み込まれており、量子化の設定は別途必要ありません。

Microsoft ONNX Runtime の詳細は、<https://microsoft.github.io/onnxruntime/> を参照してください。

Vitis AI と ONNX Runtime の統合に関する Vitis AI のチュートリアルとインストール ガイドは、Vitis AI の GitHub リポジトリ (<https://github.com/Xilinx/Vitis-AI/tree/master/external/onnxruntime>) にあります。

モデルのプロファイリング

この章では、Vitis™ AI 開発キットに含まれるユーティリティ ツールについて説明します。Vitis AI プロファイラーを除くこれらのツールの大半は、エッジ DPU にのみ使用できます。Vitis AI プロファイラーは、AI アプリケーションのプロファイリングと可視化に使用される一連の VART ベース ツールです。このキットには 5 つのツールがあり、それぞれ DPU のデバッグ、性能プロファイリング、DPU ランタイム モード操作、および DPU コンフィギュレーション ファイルの生成に使用可能です。これらのツールを組み合わせて使用すると、DPU のデバッグと性能プロファイリングを個別に実行できます。

Vitis AI プロファイラー

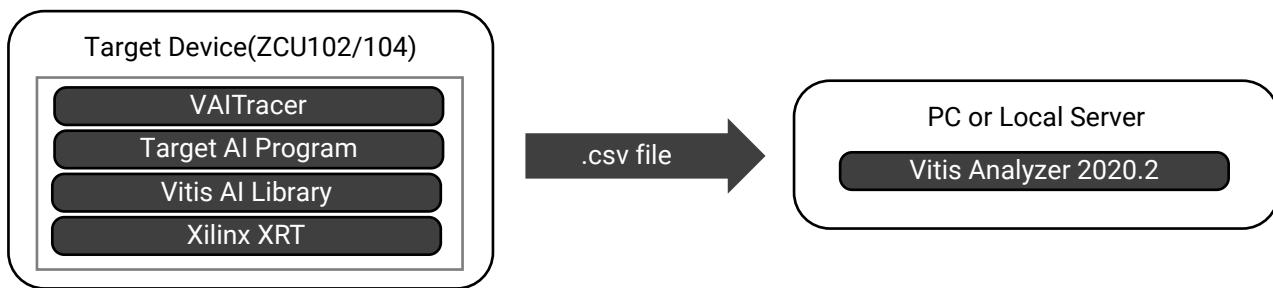
Vitis AI プロファイラーは、Vitis AI 用のオールインワンのプロファイリング ソリューションです。アプリケーション レベルのツールであり、VART ベースで AI アプリケーションのプロファイリングと可視化を実行します。AI アプリケーションには、ハードウェアで実行されるコンポーネント（通常 DPU で実行されるニューラル ネットワーク計算など）と、C/C++ コードで実装される関数として CPU で実行されるコンポーネント（画像の前処理など）があります。このツールは、これらの各種コンポーネントすべての実行ステータスをまとめて表示します。

- 簡単に使用でき、ユーザー コードの変更やプログラムの再コンパイルは不要です。
- システム パフォーマンスのボトルネックを可視化します。
- 各種演算ユニット (CPU/GPU) の実行ステータスを表示します。

Vitis AI プロファイラーのアーキテクチャ

次の図に、Vitis AI プロファイラーのアーキテクチャを示します。

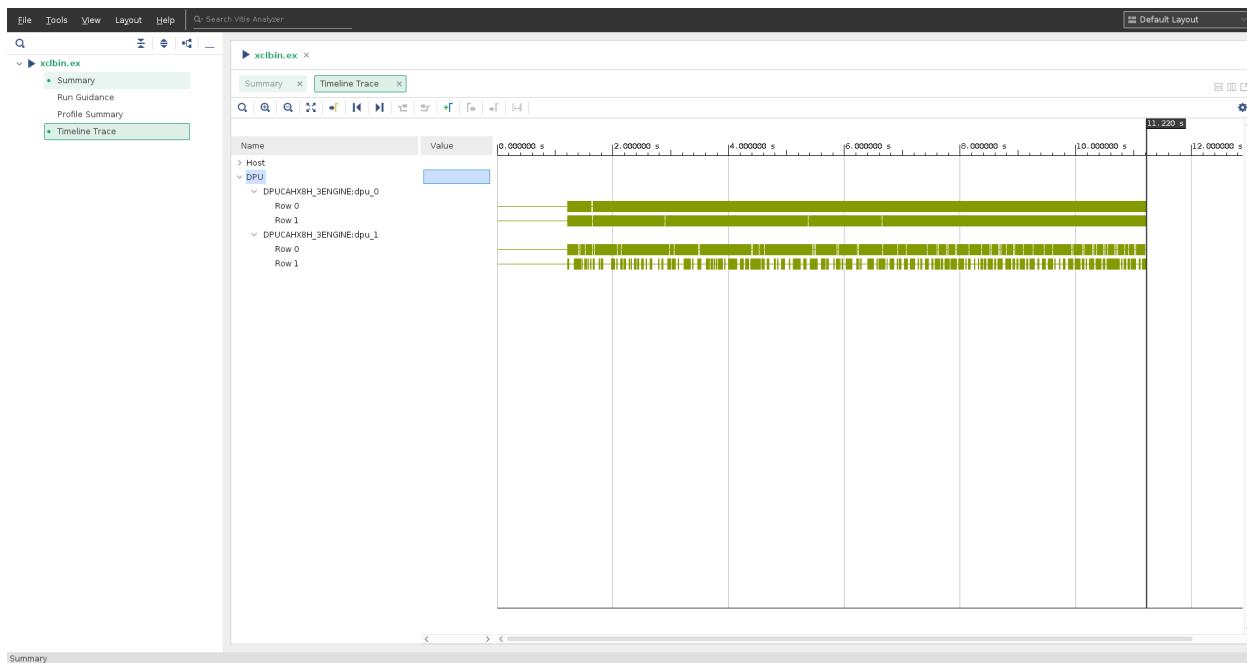
図 29: Vitis AI プロファイラーのアーキテクチャ



X24604-120420

Vitis AI プロファイラー GUI の概要

図 30: Vitis AI プロファイラー GUI の概要

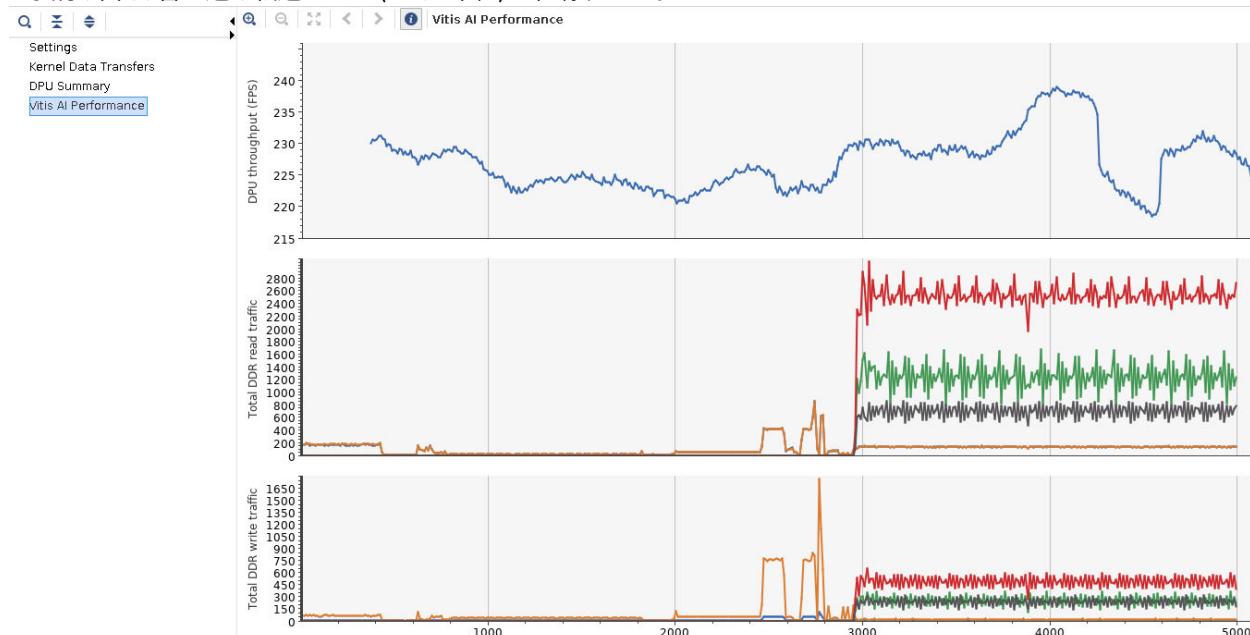


- DPU の一覧: 各カーネルの実行回数と最小/平均/最大時間 (ms) の表。

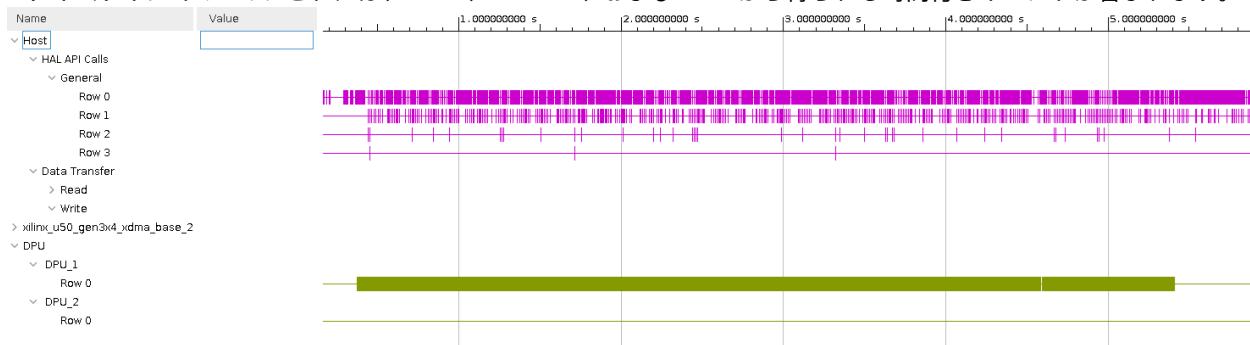
DPU Summary

Kernel	Compute Unit	Runs	Min Time (ms)	Avg Time (ms)	Max Time (ms)
subgraph_conv1	DPUCAHX8H_3ENGINE:dpu_0	175	0.550	0.685	0.590
subgraph_res2a_branch2a	DPUCAHX8H_3ENGINE:dpu_0	175	0.254	0.362	0.294
subgraph_res2a_branch2b	DPUCAHX8H_3ENGINE:dpu_0	175	0.415	0.539	0.456
subgraph_res2a_branch2c	DPUCAHX8H_3ENGINE:dpu_0	175	0.469	0.613	0.505
subgraph_res2a	DPUCAHX8H_3ENGINE:dpu_0	175	0.641	0.754	0.679
subgraph_res2b_branch2a	DPUCAHX8H_3ENGINE:dpu_0	175	0.373	0.505	0.411
subgraph_res2b_branch2b	DPUCAHX8H_3ENGINE:dpu_0	175	0.417	0.581	0.453
subgraph_res2b	DPUCAHX8H_3ENGINE:dpu_0	175	0.637	0.750	0.676
subgraph_res2c_branch2a	DPUCAHX8H_3ENGINE:dpu_0	175	0.372	0.501	0.414
subgraph_res2c_branch2b	DPUCAHX8H_3ENGINE:dpu_0	175	0.418	0.545	0.455
subgraph_res2c	DPUCAHX8H_3ENGINE:dpu_0	175	0.683	0.794	0.724
subgraph_res3a_branch2a	DPUCAHX8H_3ENGINE:dpu_0	175	0.230	0.339	0.271
subgraph_res3a_branch2b	DPUCAHX8H_3ENGINE:dpu_0	175	0.429	0.591	0.463
subgraph_res3a_branch2c	DPUCAHX8H_3ENGINE:dpu_0	175	0.336	0.447	0.369
subgraph_res3a	DPUCAHX8H_3ENGINE:dpu_0	175	0.501	0.669	0.536
subgraph_res3b_branch2a	DPUCAHX8H_3ENGINE:dpu_0	175	0.327	0.476	0.361
subgraph_res3b_branch2b	DPUCAHX8H_3ENGINE:dpu_0	175	0.425	1.122	0.463
subgraph_res3b	DPUCAHX8H_3ENGINE:dpu_0	175	0.465	0.612	0.501
subgraph_res3c_branch2a	DPUCAHX8H_3ENGINE:dpu_0	175	0.325	0.422	0.359
subgraph_res3c_branch2b	DPUCAHX8H_3ENGINE:dpu_0	175	0.426	0.548	0.461
subgraph_res3c	DPUCAHX8H_3ENGINE:dpu_0	175	0.460	0.600	0.501
subgraph_res3d_branch2a	DPUCAHX8H_3ENGINE:dpu_0	175	0.330	0.451	0.365
subgraph_res3d_branch2b	DPUCAHX8H_3ENGINE:dpu_0	175	0.430	0.591	0.463
subgraph_res3d	DPUCAHX8H_3ENGINE:dpu_0	175	0.567	0.734	0.602
subgraph_res4a_branch2a	DPUCAHX8H_3ENGINE:dpu_0	175	0.267	0.403	0.318
subgraph_res4a_branch2b	DPUCAHX8H_3ENGINE:dpu_0	175	0.408	0.580	0.448
subgraph_res4a_branch2c	DPUCAHX8H_3ENGINE:dpu_0	175	0.307	0.458	0.342
subgraph_res4a	DPUCAHX8H_3ENGINE:dpu_0	175	0.513	0.651	0.560
subgraph_res4b_branch2a	DPUCAHX8H_3ENGINE:dpu_0	175	0.301	0.414	0.341

- DPU のスループットと DDR 転送レート: アプリケーションの実行中にサンプリングされた、実現された FPS および読み出し/書き込み転送レート (MB/s 単位) の直線グラフ。



- タイムライントレース: これには、VART、HAL API、および DPU から得られる時間付きイベントが含まれます。



注記:

1. Vitis AI 1.3 およびそれ以降のリリースでは、Vitis アナライザーが vaitrace のデフォルト GUI になります。
2. 従来のウェブベースの Vitis AI プロファイラーは、Vitis AI 1.3 ではエッジ デバイス用に動作します (Zynq UltraScale+ MPSoC)。 詳細は、Vitis AI プロファイラー v1.2 の README を参照してください。

Vitis AI プロファイラーの使用

システム要件

- ハードウェア:
 - Zynq UltraScale+ MPSoC (DPUCZD シリーズ) をサポート
 - Alveo アクセラレータ カード (DPUCAH シリーズ) をサポート
- ソフトウェア:
 - VART v1.3+ をサポート

Vitis AI プロファイラーのインストール

1. Zynq UltraScale+ MPSoC PetaLinux プラットフォームで vaitrace 用のデバッグ環境を準備します。
 - a. `petalinux-config -c kernel` を実行することにより、PetaLinux のコンフィギュレーションと構築を実行します。
 - b. Linux カーネルに対して次の設定を有効にします。
 - 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. `petalinux-config -c rootfs` を実行し、root-fs に対して次の設定を有効にします。
 - user-packages ---> modules ---> [*] packagegroup-petalinux-self-hosted
 - d. `petalinux-build` を実行します。

2. vaitrace をインストールします。vaitrace は VART ランタイムに統合されます。VART ランタイムがインストールされている場合、vaitrace は /usr/bin/vaitrace にインストールされます。

vaitrace を使用して簡単なトレースを開始

次の例では、VART ResNet50 サンプルを使用します。

1. Vitis AI をダウンロードして設定します。
2. テストとトレースを開始します。
 - C++ プログラムの場合、次のようにテスト コマンドの前に vaitrace を追加します。

```
# cd ~/Vitis_AI/examples/VART/samples/resnet50
# vaitrace ./resnet50 /usr/share/vitis_ai_library/models/resnet50/
resnet50.xmodel
```

- Python プログラムの場合、次のように python インターフェリター コマンドに -m vaitrace_py を追加します。

```
# cd ~/Vitis_AI/examples/VART/samples/resnet50_mt_py
# python3 -m vaitrace_py ./resnet50.py 2 /usr/share/vitis_ai_library/
models/resnet50/resnet50.xmodel
```

vaitrace と XRT は、作業ディレクトリにいくつかのファイルを生成します。

3. すべての .csv ファイルと xclbin.ex.run_summary をユーザーのシステムにコピーします。vitis_analyzer 2020.2 以上を使用して、xclbin.ex.run_summary を開くことができます。
 - コマンド ラインを使用する場合は、# vitis_analyzer xclbin.ex.run_summary を実行します。
 - GUI を使用する場合は、[File]→[Open Summary]→[xclbin.ex.run_summary] をクリックします。

Vitis アナライザーの詳細は、「Vitis Analyzer の使用」を参照してください。

VAI トレースの使用法

コマンド ラインの使用法

```
# vaitrace --help
usage: Xilinx Vitis AI Trace [-h] [-c [CONFIG]] [-d] [-o [TRACESAVETO]] [-t
[TIMEOUT]] [-v]

      cmd                  Command to be traced
      -b                  Bypass mode, just run command and by pass vaitrace,
for debug use
      -c [CONFIG]          Specify the configuration file
      -o [TRACESAVETO]    Save trace file to
      -t [TIMEOUT]         Tracing time limitation, default value is 30 for vitis
analyzer format, and 5 for .xat format
      --va                Generate trace data for Vitis Analyzer
      --xat               Generate trace data in .xat, for the legacy web based
Vitis-AI Profiler, only available for Zynq MPSoC devices
```

次に、使用頻度の高い重要な引数を示します。

- cmd: cmd は、トレースされる Vitis AI の実行可能なプログラムです。

- -t: [cmd] の起動から始まるトレース タイム(秒)を制御します。デフォルト値は 30 です。つまり、vaitrace に -t が指定されていない場合、[cmd] を 30 秒間実行した後にトレースは停止します。[cmd] の実行は通常どおり続きますが、トレース データの収集は停止します。一度にトレースする画像は約 50 ~ 100 個にすることを推奨します。50 個より少ないと十分な統計情報が得られないことがあります、100 個より多いとシステムの実行速度が大幅に低下します。
- -c: カスタム オプションを指定してトレースを開始するには、JSON コンフィギュレーション ファイルにオプションを書き込み、そのコンフィギュレーション ファイルを -c で指定します。コンフィギュレーション ファイルの詳細は、次のセクションで説明します。

その他の引数は、デバッグに使用されます。

コンフィギュレーション

コンフィギュレーション ファイルを使用して vaitrace のトレース オプションを記録することを推奨します。
vaitrace -c trace_cfg.json を使用して、保存したコンフィギュレーションでトレースを開始できます。

コンフィギュレーションの優先順位: [コンフィギュレーション ファイル] → [コマンド ライン] → [デフォルト]

次に、vaitrace コンフィギュレーション ファイルの例を示します。

```
{
  "options": {
    "runmode": "normal",
    "cmd": "/usr/share/vitis-ai-library/sample/classification/test_jpeg_classification resnet50 sample.jpg",
    "output": "./trace_resnet50.xat",
    "timeout": 3
  },
  "trace": {
    "enable_trace_list": [ "vitis-ai-library", "vart", "custom" ],
    "trace_custom": []
  }
}
```

表 29: コンフィギュレーション ファイルの内容

キー名	形式	説明
オプション	オブジェクト	vaitrace のオプション
cmd	文字列	コマンド ライン引数 cmd と同じ
	文字列	コマンド ライン引数 -o と同じ
	整数	コマンド ライン引数 -t と同じ
	文字列	xmodel の実行モード制御。debug または normal。runmode == debug の場合、VART はこれを用いてデバッグ モードで xmodel の実行を制御し、xmodel の細粒度のプロファイリングが可能になります。
trace	オブジェクト	
	リスト	イネーブルにする内蔵トレース機能のリスト。利用可能な値は、vitis-ai-library、vart、opencv、custom です (custom は trace_custom リスト内の関数を示す)。

表 29: コンフィギュレーション ファイルの内容 (続き)

キー名	形式	説明
trace_custom	リスト	ユーザーによってインプリメントされる、トレースされる関数のリスト。関数の名前用に、名前空間がサポートされています。カスタムトレース関数の使用例は、このガイドの後半に記載されています。

DPU のプロファイリング例

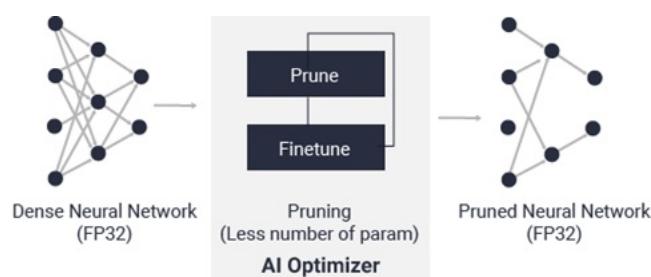
GitHub の「Vitis-AI-Profiler」のページの「Vitis-AI Profiler」に、より高度な DPU プロファイリングの例が掲載されています。

モデルの最適化

注記: モデルの最適化はオプションの手順です。

Vitis AI オプティマイザーは、ニューラル ネットワーク モデルの最適化機能を提供します。現在、Vitis AI オプティマイザーには Vitis AI プルーナー (VAI プルーナー) と呼ばれる 1 つのツールのみが含まれています。このツールは、ニューラル ネットワーク内の冗長な接続をプルーニングし、必要な動作を全体的に削減します。VAI プルーナーで生成したプルーニング済みモデルは、VAI クオンタイザーで量子化して FPGA 上で運用できます。

図 31: Vitis AI オプティマイザー



VAI プルーナーは、4 つの深層学習フレームワーク (TensorFlow、PyTorch、Caffe、Darknet) をサポートします。各フレームワークに対応するツール名は、それぞれ vai_p_tensorflow、vai_p_pytorch、vai_p_caffe、vai_p_darknet であり、中間の「p」はプルーニングを示します。

詳細は、『Vitis AI オプティマイザー ユーザー ガイド』 ([UG1333](#)) を参照してください。

Vitis AI オプティマイザーを実行するには、コマーシャル ライセンスが必要です。詳細は、ザイリンクス販売代理店までお問い合わせください。

ML フレームワークを使用したサブグラフの高速化

パーティショニングとは、FPGA とホストの間でモデルの推論実行を分割するプロセスです。FPGA でサポートされていないレイヤーを含むモデルを実行する場合に、このパーティショニングが必要です。また、目標を達成するために、異なる計算グラフのパーティショニングや実行を試したり、デバッグする際にも役立ちます。

注記: この機能は現在、DPUCADX8G を使用する Alveo™ U200/U250 でのみ利用できます。

TensorFlow でファンクショナル API 呼び出しを分割

グラフの分割には、次のような一般的なフローがあります。

1. パーティション クラスを作成/初期化します。

```
from vai.dpuv1.rt.xdnn_rt_tf import TFxdnnRT  
xdnnTF = TFxdnnRT(args)
```

2. 分割したグラフを読み込みます。

```
graph = xdnnTF.load_partitioned_graph()
```

3. オリジナル グラフが読み込まれているかのように、前処理と後処理を適用します。

パーティショナー API

パーティショナーの主な入力引数(例: パーティショニング使用フローの項目 1 の引数)は次のとおりです。

- Networkfile: tf.Graph、tf.GraphDef、またはネットワーク ファイルへのパス
- loadmode: ネットワーク ファイルの保存プロトコル。サポートされている形式 [pb (default), chkpt, txt, savedmodel]
- quant_cfgfile: DPUCADX8G 量子化ファイル
- batch_sz: 推論バッチ サイズ。デフォルト値は 1 です。
- startnode: FPGA パーティションの開始ノードのリスト(オプション。デフォルトはすべてのプレースフォルダー)
- finalnode: FPGA パーティションの最終ノードのリスト(オプション。デフォルトはすべてのシンク ノード)

パーティショニングの手順

1. オリジナル グラフを読み込む

パーティショナーを使用して、凍結された `tf.Graph`、`tf.GraphDef`、またはネットワーク ファイル/フォルダーへのパスを処理できます。`.pb` ファイルが提供されている場合は、グラフが適切に凍結されているはずです。その他のオプションには、`tf.train.Saver` および `tf.saved_model` を使用するモデル ストアがあります。

2. パーティショニング

この手順では、`startnode` および `finalnode` で指定されたサブグラフが FPGA アクセラレーション用に分析されます。これは複数フェーズで実行されます。

- すべてのグラフノードは、2 つの方法のいずれかを使用して、(FPGA で) サポートされているセットとサポートされていないセットに分割されます。デフォルト (`compilerFunc='SPECULATIVE'`) の方法では、ハードウェア オペレーション ツリーの概算を使用します。2 番目の方法 (`compilerFunc= 'DEFINITIVE'`) では、ハードウェア コンパイラを使用します。後者の方がより正確で、指定されたオプションに基づいて複雑な最適化方法を処理できますが、プロセスを完了するのにかなりの時間を要します。
- 隣接するサポートされているノードとサポートされていないノードは、(ファイングレインで) 接続されたコンポーネントにマージされます。
- サポートされているパーティションは、DAG プロパティを維持しながら、最大に接続されたコンポーネントにマージされます。
- サポートされている各パーティションは、ハードウェア コンパイラを使用して(再び) コンパイルされ、ランタイム コード、量子化情報、および関連するモデル パラメーターが作成されます。
- サポートされる各パーティションのサブグラフは、視覚化およびデバッグ用に保存されます。
- サポートされる各サブグラフは、FPGA でそのサブグラフを高速化するために必要なすべての Python 関数呼び出しを含む `tf.py_func` ノードに置き換えられます (`fpga_func_<partition_id>` という命名規則を使用)。

3. 変更されたグラフを凍結する

変更されたグラフは凍結され、「`-fpga`」という接尾辞が付けられて保存されます。

4. TensorFlow でネイティブに実行する

変更されたグラフは、「`load_partitioned_graph`」のパーティショナー クラスを使用して読み込み可能です。変更されたグラフは、デフォルトの TensorFlow グラフを置き換えて、オリジナル グラフと同じように使用できます。

実用メモ

コンパイラの最適化は、位置引数またはオプション引数を介してパーティショナー クラス `TFxdnnRT` に適用可能なコンパイラ引数を渡すことで変更できます。モデルが適切に凍結されていない場合、コンパイラはバッチ正規化などの一部の動作の最適化に失敗する可能性があります。

`startnode` と `finalnode` のセットは頂点分離である必要があります。つまり、`startnode` または `finalnode` を削除すると、グラフは 2 つの異なる接続コンポーネントに切り離されます (`startnode` がグラフ プレスホールダーのサブセットである場合を除く)。

可能な限り、单一のマクロレイヤーとして実行されるレイヤー間のカットノードを指定しないでください。たとえば、`Conv(x) -> BiasAdd(x)` で、`Conv(x)` を `BiasAdd(x)` とは異なる FPGA パーティションに配置すると、結果として最適でない性能(スループット、レイテンシ、および精度)になります。

パーティショナーの初期化には、FPGA の実行可能コードを作成できるようする `quant_cfgfile` が必要です。FPGA の実行が意図されない場合、`quant_cfgfile="IGNORE"` を設定することで、この要件を回避できます。

Caffe のパーティショニング サポート

ザイリンクスは、Caffe グラフを自動的に分割するために Caffe パッケージの機能を強化しました。この機能は、ネットワーク内の FPGA 実行可能レイヤーを分離し、推論で使用される新しい prototxt を生成します。サブグラフ カッターは、カスタム Python レイヤーを作成して、FPGA 上での高速化を図ります。次のコード スニペットを参照してください。

```
from vai.dpuv1.rt.scripts.framework.caffe.xfdnn_subgraph \
    import CaffeCutter as xfdnnCutter
def Cut(prototxt):

    cutter = xfdnnCutter(
        inproto="quantize_results/deploy.prototxt",
        trainproto=prototxt,
        outproto="xfdnn_auto_cut_deploy.prototxt",
        outtrainproto="xfdnn_auto_cut_train_val.prototxt",
        cutAfter="data",
        xclbin=XCLBIN,
        netcfg="work/compiler.json",
        quantizecfg="work/quantizer.json",
        weights="work/deploy caffemodel_data.h5"
    )
    cutter.cut()
#cutting and generating a partitioned graph auto-cut-deploy.prototxt
Cut(prototxt)
```

Cut(prototxt)

前の手順で生成された `auto_cut_deploy.prototxt` には、推論を実行するための完全な情報が含まれています。次に例を示します。

- ノートブックの実行: ノートブックのこれらの手順について理解するため、\$VAI_ALVEO_ROOT/notebooks から入手できる 2 つのサンプル ノートブック (画像検出と画像分類) があります。
- スクリプトの実行: デフォルト設定でモデルを実行するために使用できる Python スクリプトがあります。次のコマンドを使用して実行できます。
 - PreparePhase: Python

```
$VAI_ALVEO_ROOT/examples/caffe/run.py --prototxt <example prototxt> --
caffemodel <example caffemodel> --prepare
```

- prototxt: モデルの prototxt へのパス
 - caffemodel: モデルの caffemodel へのパス
 - output_dir: 量子化、コンパイラ、subgraph_cut ファイルを保存するパス
 - qtest_iter: 量子化をテストするための反復回数
 - qcalib_iter: 量子化に使用されるキャリブレーションの反復回数
- 検証フェーズ: Python

```
$VAI_ALVEO_ROOT/examples/caffe/run.py -validate
```

- output_dir: 準備フェーズで output_dir を指定する場合、準備フェーズで生成されたファイルを使用するため、同じ引数と値を使用。
- numBatches: 推論のテストに使用できるバッチの数。

カスタム プラットフォームへの DPU の統合

カスタム Vitis プラットフォームに DPU を統合して、Vitis™ ソフトウェア プラットフォームで AI アプリケーションを実行できます。コンパイル済みのプラットフォームがいくつか用意されており、ザイリンクス Vitis エンベデッド プラットフォーム ダウンロードからダウンロードできます。カスタム プラットフォームを作成する場合は、『Vitis 統合ソフトウェア プラットフォームの資料』 ([UG1416](#)) を参照してください。

DPU の統合を容易にするため、ザイリンクスは、性能とリソース使用率の要件を満たすさまざまなパラメーターで DPU を設定できる DPU TRD を提供しています。詳細は、『Zynq DPU v3.1 IP 製品ガイド』 (PG338: [英語版](#)、[日本語版](#)) および [Vitis DPU TRD フロー](#) を参照してください。

ハードウェア側では、Vitis ソフトウェア プラットフォームは DPU を RTL カーネルとして統合します。これには、clk と clk2x の 2 つのクロックが必要です。また、1 つの割り込みが必要です。DPU は複数の AXI HP インターフェイスを必要とすることがあります。

ソフトウェア側では、Vitis ソフトウェア プラットフォームは XRT および ZOCL パッケージを提供する必要があります。ホスト アプリケーションは、XRT OpenCL™ API を使用してカーネルを制御します。Vitis AI ランタイムは、XRT を使用して DPU を制御できます。ZOCL は、アクセラレーション カーネルにアクセスするカーネル モジュールです。ZOCL が動作するには、デバイスツリー ノードが追加されている必要があります。

詳細は、[Vitis AI プラットフォームの作成 チュートリアル](#)を参照してください。

DPU の統合に Vivado® ツールを使用する場合は、[DPU TRD Vivado フロー](#) を参照してください。

Alveo™ アクセラレータ カードやその他のエンベデッド以外のプラットフォームに DPU を統合する場合は、xilinx_ai_prod_mkt@xilinx.com にお問い合わせください。

Vitis AI プログラミング インターフェイス

この付録では、VART プログラミング インターフェイスで提供されるすべてのプログラミング API について説明します。

VART API

C++ API

クラス 1

クラス名は `vart::Runner` です。次の表に、`vitis::vart::Runner` クラスに定義されているすべての関数を示します。

表 30: 関数クリックリファレンス

戻り値のタイプ	名前	引数
<code>std::unique_ptr<Runner></code>	<code>create_runner</code>	<code>const xir::Subgraph* subgraph</code> <code>const std::string& mode</code>
<code>std::vector<std::unique_ptr<Runner>></code>	<code>create_runner</code>	<code>const std::string& model_directory</code>
<code>std::pair<uint32_t, int></code>	<code>execute_async</code>	<code>const std::vector<TensorBuffer*>& input</code> <code>const std::vector<TensorBuffer*>& output</code>
<code>int</code>	<code>wait</code>	<code>int jobID</code> <code>int timeout</code>
<code>TensorFormat</code>	<code>get_tensor_format</code>	
<code>std::vector<const xir::Tensor*></code>	<code>get_input_tensors</code>	
<code>std::vector<const xir::Tensor*></code>	<code>get_output_tensors</code>	

クラス 2

クラス名は `vart::TensorBuffer` です。次の表に、`vart::TensorBuffer` クラスに定義されているすべての関数を示します。

表 31: 関数クイック リファレンス

戻り値のタイプ	名前	引数
location_t	get_location	
const xir::Tensor*	get_tensor	
std::pair<std::uint64_t, std::size_t>	data	const std::vector<std::int32_t> idx = {}
std::pair<uint64_t, size_t>	data_phy	const std::vector<std::int32_t> idx
void	sync_for_read	uint64_t offset、size_t size
void	sync_for_write	uint64_t offset、size_t size
void	copy_from_host	size_t batch_idx、const void* buf、size_t size、size_t offset
void	copy_to_host	size_t batch_idx、void* buf、size_t size、size_t offset
void	copy_tensor_buffer	vart::TensorBuffer* tb_from、vart::TensorBuffer* tb_to

クラス 3

クラス名は `vart::RunnerExt` です。次の表に、`vart::RunnerExt` クラスに定義されているすべての関数を示します。

表 32: 関数クイック リファレンス

戻り値のタイプ	名前	引数
std::vector<vart::TensorBuffer*>	get_inputs	
std::vector<vart::TensorBuffer*>	get_outputs	

create_runner

CPU/SIM/DPU ランナーのインスタンスをサブグラフ別に作成します。これはザイリンクス側で使用するメソッドです。

プロトタイプ

```
std::unique_ptr<Runner> create_runner(const xir::Subgraph* subgraph,
                                         const std::string& mode = "");
```

パラメーター

次の表に、`create_runner` 関数の引数を示します。

 表 33: `create_runner` の引数

タイプ	名前	説明
const xir::Subgraph*	subgraph	XIR サブグラフ
const std::string&	mode	3 つのモードをサポートしています。 ref - CPU ランナー sim - シミュレーション run - DPU ランナー

戻り値

CPU/SIM/DPU ランナーのインスタンス。

create_runner

model_directory ごとに DPU ランナーを作成します。

プロトタイプ

```
std::vector<std::unique_ptr<Runner>> create_runner(const std::string&
model_directory);
```

パラメーター

次の表に、`create_runner` 関数の引数を示します。

表 34: `create_runner` の引数

タイプ	名前	説明
const std::string&	model_directory	meta.json を格納するディレクトリ名。

戻り値

DPU ランナーのベクター。

execute_async

ランナーを実行します。ブロック関数です。

プロトタイプ

```
virtual std::pair<uint32_t, int> execute_async(
    const std::vector<TensorBuffer*>& input,
    const std::vector<TensorBuffer*>& output) = 0;
```

パラメーター

次の表に、`execute_async` 関数の引数を示します。

表 35: `execute_async` の引数

タイプ	名前	説明
const std::vector<TensorBuffer*>&	input	推論用の入力データを格納する入力テンソルバッファーのベクター。
const std::vector<TensorBuffer*>&	output	出力データで充填される出力テンソルバッファーのベクター。

戻り値

`pair<jobID, status>`。ステータス 0 は正常終了、その他の値はカスタマイズされた警告またはエラー。

wait

DPU 处理の終了を待機します。ブロック関数です。

プロトタイプ

```
int wait(int jobid, int timeout)
```

パラメーター

次の表に、wait 関数の引数を示します。

表 36: wait の引数

タイプ	名前	説明
int	jobid	ジョブ ID。負の値は任意の ID、その他の値は特定のジョブ ID。
int	timeout	タイムアウト。負の値は永久的にブロック、0 はノンブロック、正の値は制限 (ms) 付きブロック。

戻り値

ステータス 0 は正常終了、その他の値はカスタマイズされた警告またはエラー。

get_tensor_format

ランナーのテンソル形式を取得します。

プロトタイプ

```
TensorFormat get_tensor_format();
```

パラメーター

なし

戻り値

TensorFormat: NHWC/HCHW

get_input_tensors

ランナーのすべての入力テンソルを取得します。

プロトタイプ

```
std::vector<const xir::Tensor*> get_input_tensors()
```

パラメーター

なし

戻り値

すべての入力テンソル。入力テンソルへの生ポインターのベクター。

get_output_tensors

ランナーのすべての出力テンソルを取得します。

プロトタイプ

```
std::vector<const xir::Tensor*> get_output_tensors()
```

パラメーター

なし

戻り値

すべての出力テンソル。出力テンソルへの生ポインターのベクター。

get_location

テンソルバッファーの位置を取得します。

プロトタイプ

```
location_t get_location();
```

パラメーター

なし

戻り値

テンソルバッファーの位置、location_t enum 型の値。

次の表に、location_t enum 型を示します。

表 37: location_t enum 型

名前	値	説明
HOST_VIRT	0	ホストのみアクセス可能。
HOST_PHY	1	ホストとデバイスに共有される連続物理メモリ。

表 37: **location_t enum** 型 (続き)

名前	値	説明
DEVICE_0	2	device_* のみアクセス可能。
DEVICE_1	3	
DEVICE_2	4	
DEVICE_3	5	
DEVICE_4	6	
DEVICE_5	7	
DEVICE_6	8	
DEVICE_7	9	

get_tensor

TensorBuffer のテンソルを取得します。

プロトタイプ

```
const xir::Tensor* get_tensor()
```

パラメーター

なし

戻り値

テンソルへのポインター。

data

インデックスのデータ アドレスと残されたアクセス可能なデータ サイズを取得します。

プロトタイプ

```
std::pair<std::uint64_t, std::size_t> data(const std::vector<std::int32_t>
idx = {});
```

パラメーター

次の表に、data 関数の引数を示します。

 表 38: **data の引数**

タイプ	名前	説明
const std::vector<std::int32_t>	idx	アクセスされるデータのインデックス。 次元はテンソルの形状と同じです。

戻り値

インデックスのデータ アドレスと残されたアクセス可能なデータ サイズ (バイト単位) のペア。

data_phy

インデックスのデータ物理アドレスと残されたアクセス可能なデータ サイズを取得します。

プロトタイプ

```
std::pair<uint64_t, size_t> data_phy(const std::vector<std::int32_t> idx);
```

パラメーター

次の表に、`data_phy` 関数の引数を示します。

表 39: `data_phy` の引数

タイプ	名前	説明
const std::vector<std::int32_t>	idx	アクセスされるデータのインデックス。 次元はテンソルの形状と同じです。

戻り値

インデックスのデータ物理アドレスと残されたアクセス可能なデータ サイズ (バイト単位) のペア。

sync_for_read

読み出しの前に、読み出し用キャッシュを無効化します。`get_location()` が DEVICE_ONLY または HOST_VIRT を返す場合、動作なし (NOP) です。

プロトタイプ

```
void sync_for_read(uint64_t offset, size_t size) {};
```

パラメーター

次の表に、`sync_for_read` 関数の引数を示します。

表 40: `sync_for_read` の引数

タイプ	名前	説明
uint64_t	offset	開始オフセット アドレス
size_t	size	データ サイズ

戻り値

なし

sync_for_write

書き込みの後に、書き込み用キャッシュをフラッシュします。get_location() が DEVICE_ONLY または HOST_VIRT を返す場合、動作なし (NOP) です。

プロトタイプ

```
void sync_for_write (uint64_t offset, size_t size) {};
```

パラメーター

次の表に、sync_for_write 関数の引数を示します。

表 41: sync_for_write の引数

タイプ	名前	説明
uint64_t	offset	開始オフセット アドレス
size_t	size	データ サイズ

戻り値

なし

copy_from_host

ソース バッファーからデータをコピーします。

プロトタイプ

```
void copy_from_host(size_t batch_idx, const void* buf, size_t size, size_t offset);
```

パラメーター

次の表に、copy_from_host 関数の引数を示します。

表 42: copy_from_host Arguments

タイプ	名前	説明
size_t	batch_idx	バッチ インデックス
const void*	buf	ソース バッファー開始アドレス
size_t	size	コピーするデータのサイズ
size_t	offset	コピーする開始オフセット

戻り値

なし

copy_to_host

デスティネーション バッファーにデータをコピーします。

プロトタイプ

```
void copy_to_host(size_t batch_idx, void* buf, size_t size, size_t offset);
```

パラメーター

次の表に、`copy_to_host` 関数の引数を示します。

表 43: `copy_to_host` の引数

タイプ	名前	説明
size_t	batch_idx	バッチ インデックス
void*	buf	デスティネーション バッファー開始アドレス
size_t	size	コピーするデータのサイズ
size_t	offset	コピーする開始オフセット

戻り値

なし

copy_tensor_buffer

TensorBuffer をコピーします。

プロトタイプ

```
static void copy_tensor_buffer(vart::TensorBuffer* tb_from,
vart::TensorBuffer* tb_to);
```

パラメーター

次の表に、`copy_tensor_buffer` 関数の引数を示します。

表 44: `copy_tensor_buffer` の引数

タイプ	名前	説明
vart::TensorBuffer*	tb_from	ソース TensorBuffer
vart::TensorBuffer*	tb_to	デスティネーション TensorBuffer

戻り値

なし

get_inputs

RunnerExt のすべての入力 TensorBuffer を取得します。

プロトタイプ

```
std::vector<vart::TensorBuffer*> get_inputs();
```

パラメーター

なし

戻り値

すべての TensorBuffer。入力 TensorBuffer への生ポインターのベクター。

get_outputs

RunnerExt のすべての出力 TensorBuffer を取得します。

プロトタイプ

```
std::vector<vart::TensorBuffer*> get_outputs();
```

パラメーター

なし

戻り値

すべての出力 TensorBuffer。出力 TensorBuffer への生ポインターのベクター。

Python API

クラス 1

クラス名は `vart.Runner` です。次の表に、`vart.Runner` クラスに定義されているすべての関数を示します。

表 45: 関数クリックリファレンス

タイプ	名前	引数
<code>vart.Runner</code>	create_runner	<code>xir.Subgraph subgraph</code> <code>string mode</code>
<code>List[xir.Tensor]</code>	get_input_tensors	
<code>List[xir.Tensor]</code>	get_output_tensors	
<code>tuple[uint32, int]</code>	execute_async	<code>List[vart.TensorBuffer] inputs</code> <code>List[vart.TensorBuffer] outputs</code> 注記: パッファー プロトコルを備えた <code>vart.TensorBuffer</code> 。
<code>int</code>	wait	<code>tuple[uint32, int] jobID</code>

クラス 2

クラス名は `vart.RunnerExt` です。次の表に、`vart.RunnerExt` クラスに定義されているすべての関数を示します。

`vart.RunnerExt` は `vart.Runner` から拡張された関数です。

表 46: 関数クイック リファレンス

タイプ	名前	引数
<code>vart.RunnerExt</code>	<code>create_runner</code>	<code>xir.Subgraph subgraph</code> <code>string mode</code>
<code>List[vart.TensorBuffer]</code>	<code>get_inputs</code>	
<code>List[vart.TensorBuffer]</code>	<code>get_outputs</code>	

`create_runner`

DPU ランナーのインスタンスをサブグラフ別に作成します。ファクトリ関数です。

プロトタイプ

```
vart.Runner create_runner(xir.Subgraph subgraph, String mode)
```

パラメーター

次の表に、`create_runner` 関数の引数を示します。

表 47: `create_runner` の引数

タイプ	名前	説明
<code>xir.Subgraph</code>	<code>subgraph</code>	XIR サブグラフ
文字列	<code>mode</code>	run - DPU ランナー

戻り値

DPU ランナーのインスタンス。

`execute_async`

ランナーを実行します。ブロック関数です。

プロトタイプ

```
tuple[uint32_t, int] execute_async(
    List[vart.TensorBuffer] inputs,
    List[vart.TensorBuffer] outputs)
```

注記: バッファー プロトコルを備えた `vart.TensorBuffer`。

パラメーター

次の表に、`execute_async` 関数の引数を示します。

表 48: `execute_async` の引数

タイプ	名前	説明
List[vart.TensorBuffer]	入力	推論用の入力データを格納する vart.TensorBuffer のリスト。
List[vart.TensorBuffer]	出力	出力データで充填される vart.TensorBuffer のリスト。

戻り値

`tuple[jobID, status]`。ステータス 0 は正常終了、その他の値はカスタマイズされた警告またはエラー。

wait

DPU 処理の終了を待機します。ブロック関数です。

プロトタイプ

```
int wait(tuple<uint32_t, int> jobid)
```

パラメーター

次の表に、`wait` 関数の引数を示します。

表 49: `wait` の引数

タイプ	名前	説明
tuple<uint32_t, int>	jobid	ジョブ ID

戻り値

ステータス 0 は正常終了、その他の値はカスタマイズされた警告またはエラー。

get_input_tensors

ランナーのすべての入力テンソルを取得します。

プロトタイプ

```
List[xir.Tensor] get_input_tensors()
```

パラメーター

なし

戻り値

すべての入力テンソル。入力テンソルへの生ポインターのベクター。

get_output_tensors

ランナーのすべての出力テンソルを取得します。

プロトタイプ

```
List[xir.Tensor] get_output_tensors()
```

パラメーター

なし

戻り値

すべての出力テンソル。出力テンソルへの生ポインターのベクター。

get_inputs

RunnerExt のすべての入力 TensorBuffer を取得します。

プロトタイプ

```
List[vart.TensorBuffer] get_inputs()
```

パラメーター

なし

戻り値

すべての入力 TensorBuffer。入力 TensorBuffer への生ポインターのベクター。

get_outputs

RunnerExt のすべての出力 TensorBuffer を取得します。

プロトタイプ

```
List[vart.TensorBuffer] get_outputs()
```

パラメーター

なし

戻り値

すべての出力 TensorBuffer。出力 TensorBuffer への生ポインターのベクター。

レガシ DNNDK

DNNDK (Deep Neural Network Development Kit) は、深層学習プロセッシング ユニット (DPU) 向けのフルスタック 深層学習 SDK です。プルーニング、量子化、コンパイル、最適化、およびランタイムをサポートしており、ディープ ニューラル ネットワーク推論アプリケーション開発にワンストップで対応します。主な特長と機能は次のとおりです。

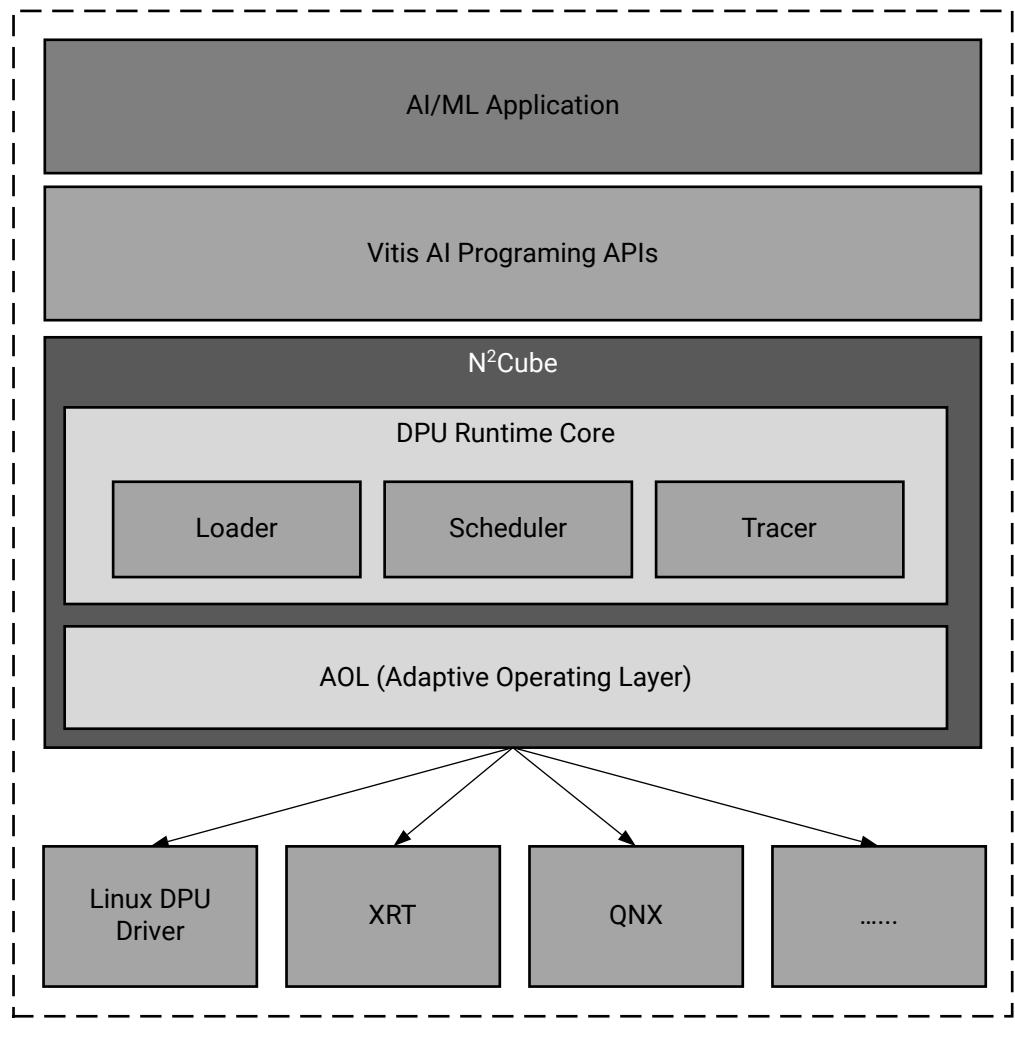
DNNDK N2Cube ランタイム

次の図に、エッジ DPUCZDX8G、レガシ DNNDK ランタイム フレームワーク (N2Cube) を示します。Vitis AI リリースでは、N2Cube は XRT をベースにしています。従来の Vivado® ベース DPU は、基礎となる Linux DPU ドライバー (XRT の代わり) を使用して DPU のスケジューリングやリソースを管理します。

N2Cube は、Vitis AI v1.2 リリースからオープンソースとして利用可能になりました。詳細は、[Vitis AI リポジトリ](#) の tools/Vitis-AI-Runtime/DNNDK ディレクトリを参照してください。

注記: N2Cube は、今後のリリースで廃止されるため、新規アプリケーションやプロジェクトには、VART を代わりに使用してください。

図 32: レガシ MPSoC のランタイム スタック



X24602-091620

N2Cube は、エッジ シナリオのさまざまな要件に柔軟に対応するために、包括的な C++/Python プログラミング インターフェイスを提供します。エッジ DPU 向けの高度なプログラミングの詳細は、[DNNDK プログラミング API](#) を参照してください。N2Cube の特長は次のとおりです。

- マルチスレッドおよびマルチプロセス DPU アプリケーションの運用をサポート。
- 複数モデルの同時実行および動作時におけるオーバーヘッドのない動的切り替えをサポート。
- ワークロード バランスを改善するための自動化された DPU マルチコア スケジューリング。
- 動作時に DPU タスクに対して DPU コアのアフィニティを動的に指定する柔軟なオプション機能。
- DPU コアのアフィニティに従いながら、優先度に基づいた DPU タスクのスケジューリング。
- マルチスレッド DPU アプリケーション内での DPU コードおよびパラメーターの共有によって最適化されるメモリ使用効率。
- QNX、VxWorks、Integrity などの POSIX 互換 OS または RTOS (リアルタイム オペレーティング システム) 環境に簡単に適応。
- DPU デバッグや性能プロファイリングのための使いやすい機能。

現在、N2Cube は Linux、ザイリンクス XRT、BlackBerry QNX RTOS の 3 つの動作環境を公式サポートしています。QNX 用の Vitis AI パッケージの取得、または N2Cube のサードパーティ製 RTOS への移植については、ザイリンクスの販売代理店へお問い合わせください。ソース コードが利用可能になれば、その他の環境に N2Cube を自由に移植できます。

DNNDK のサンプル

上位互換性を維持するために、Vitis AI は、エッジ DPUCZDX8G を使用する深層学習アプリケーション開発用に DNNDK を引き続きサポートしています。ZCU102 および ZCU104 のレガシ DNNDK の C++/Python サンプルは、<https://github.com/Xilinx/Vitis-AI/tree/master/demo/DNNDK> から入手できます。<https://github.com/Xilinx/Vitis-AI/tree/master/demo/DNNDK/README.md> のガイドラインに従って環境を設定し、これらのサンプルを評価できます。

注記: DNNDK ランタイムは、デフォルトディレクトリ `/usr/lib/` から DPU オーバーレイ .bin ファイルを読み込みます。DNNDK サンプルを実行する前に、`dpu.xclbin` が `/usr/lib/` の下にあることを確認してください。ダウンロードした ZCU102 または ZCU104 システム イメージでは、`dpu.xclbin` はデフォルトで `/usr/lib/` にコピーされます。カスタマイズしたイメージでは、ユーザーが手動で `dpu.xclbin` をコピーする必要があります。

次の表では、すべての利用可能な DNNDK サンプルについて簡単に説明しています。

表 50: DNNDK のサンプル

サンプル名	モデル	フレームワーク	注記
resnet50	ResNet50	Caffe	Vitis AI の高度な C++ API を使用する画像分類。
resnet50_mt	ResNet50	Caffe	Vitis AI の高度な C++ API を使用するマルチスレッドの画像分類。
tf_resnet50	ResNet50	TensorFlow	Vitis AI の高度な Python API を使用する画像分類。
mini_resnet_py	Mini-ResNet	TensorFlow	Vitis AI の高度な Python API を使用する画像分類。
inception_v1	Inception-v1	Caffe	Vitis AI の高度な C++ API を使用する画像分類。
inception_v1_mt	Inception-v1	Caffe	Vitis AI の高度な C++ API を使用するマルチスレッドの画像分類。
inception_v1_mt_py	Inception-v1	Caffe	Vitis AI の高度な Python API を使用するマルチスレッドの画像分類。
mobilenet	MobileNet	Caffe	Vitis AI の高度な C++ API を使用する画像分類。
mobilenet_mt	MobileNet	Caffe	Vitis AI の高度な C++ API を使用するマルチスレッドの画像分類。
face_detection	DenseBox	Caffe	Vitis AI の高度な C++ API を使用する顔検知。
pose_detection	SSD、姿勢検出	Caffe	Vitis AI の高度な C++ API を使用する姿勢検出。
video_analysis	SSD	Caffe	Vitis AI の高度な C++ API を使用するトラフィック検知。

表 50: DNNDK のサンプル (続き)

サンプル名	モデル	フレームワーク	注記
adas_detection	YOLO-v3	Caffe	Vitis AI の高度な C++ API を使用する ADAS 検知。
segmentation	FPN	Caffe	Vitis AI の高度な C++ API を使用するセマンティック セグメンテーション。
split_io	SSD	TensorFlow	Vitis AI の高度な C++ API でプログラミングされた DPU スブリット I/O メモリ モデル。
デバッグ	Inception-v1	TensorFlow	Vitis AI の高度な C++ API を使用する DPU デバッグ。
tf_yolov3_voc_py	YOLO-v3	TensorFlow	Vitis AI の高度な Python API を使用する物体検出。

評価ボード上でサンプルを実行する前に、次の表の説明に従ってイメージを準備する必要があります。

表 51: DNNDK サンプルのイメージを準備

イメージのディレクトリ	注記
vitis_ai_dnndk_samples/dataset/image500_640_480/	ImageNet データセットから複数のイメージをダウンロードし、同じ解像度 640*480 にスケーリングします。
vitis_ai_dnndk_samples2/ image_224_224/	ImageNet データセットから 1 つのイメージをダウンロードし、同じ解像度 224*224 にスケーリングします。
vitis_ai_dnndk_samples/ image_32_32/	CIFAR-10 データセット (https://www.cs.toronto.edu/~kriz/cifar.html) から複数のイメージをダウンロードします。
vitis_ai_dnndk_samples/resnet50_mt/image/	ImageNet データセットから 1 つのイメージをダウンロードします。
vitis_ai_dnndk_samples/ mobilenet_mt/image/	ImageNet データセットから 1 つのイメージをダウンロードします。
vitis_ai_dnndk_samples/ inception_v1_mt/image/	ImageNet データセットから 1 つのイメージをダウンロードします。
vitis_ai_dnndk_samples/ debugging/decent_golden/dataset/ images/	ImageNet データセットから 1 つのイメージをダウンロードし、cropped_224x224.jpg として保存します。
vitis_ai_dnndk_samples/ tf_yolov3_voc_py/image/	VOC データセット http://host.robots.ox.ac.uk/pascal/VOC/ から 1 つのイメージをダウンロードし、input.jpg として保存します。

次のセクションでは、リファレンスとして ZCU102 ボードを使用して DNNDK サンプルを実行する方法について説明します。サンプルは /workspace/mpsoc/vitis_ai_dnndk_samples ディレクトリにあるとします。各サンプルのフォルダーにある ./build.sh zcu102 スクリプトを実行することにより、Arm GCC クロス コンパイルツールチェーンによってすべてのサンプルが構築された後、ディレクトリ全体 /workspace/mpsoc/vitis_ai_dnndk_samples を ZCU102 ボードのディレクトリ (/home/root/) にコピーすることを推奨しています。1 つの DPU ハイブリッド実行ファイルを実行用として Docker コンテナーから評価ボードにコピーすることを選択できます。依存関係にあるイメージ フォルダー dataset またはビデオ フォルダー video も一緒にコピーされ、フォルダー構造も適切に保持されるようにしてください。

注記: ZCU104 ボードの DNNDK サンプルごとに ./build.sh zcu104 を実行する必要があります。

簡潔にするために、次の説明では /home/root/vitis_ai_dnndk_samples/ ディレクトリを \$dnndk_sample_base に置き換えています。

ResNet-50

Caffe ResNet-50 モデルを使用した画像分類サンプルは、\$dnndk_sample_base/resnet50 にあります。
\$dnndk_sample_base/dataset/image500_640_480 ディレクトリのイメージを読み取り、各入力イメージに対する分類結果を出力します。その後、./resnet50 コマンドを使用して起動できます。

動画解析

オブジェクト検出のサンプルは、\$dnndk_sample_base/video_analysis ディレクトリにあります。このサンプルは、動画ファイルから画像フレームを読み出し、検出した車両および歩行者にリアルタイムでアノテーションを付加します。./video_analysis video/structure.mp4 コマンドで起動します(video/structure.mp4 は入力ビデオ ファイル)。

ADAS 検出

YOLO-v3 ネットワーク モデルを使用した ADAS(先進運転支援システム) アプリケーションの物体検出サンプルは、ディレクトリ \$dnndk_sample_base/adas_detection にあります。このサンプルは、動画ファイルから画像フレームを読み出し、リアルタイムでアノテーションを付加します。./adas_detection video/adas.avi コマンドで起動します(video/adas.avi は入力ビデオ ファイル)。

セマンティック セグメンテーション

セマンティック セグメンテーション サンプルは、\$dnndk_sample_base/segmentation ディレクトリにあります。このサンプルは、動画ファイルから画像フレームを読み出し、リアルタイムでアノテーションを付加します。./segmentation video/traffic.mp4 コマンドで起動します(video/traffic.mp4 は入力ビデオ ファイル)。

Python による Inception-v1

\$dnndk_sample_base/inception_v1_mt_py に、高度な Python API を使用して開発された Inception-v1 ネットワークのマルチスレッド画像分類サンプルが含まれています。コマンド python3 inception_v1_mt.py 4 を使用すると、4 つのスレッドで実行されます。実行が完了すると、スループット(単位:fps) が表示されます。

Inception-v1 モデルは、DPU xmodel ファイルにコンパイルされた後、評価ボード上で次に示すコマンドを使用して DPU 共有ライブラリ libdpumodelinception_v1.so に変換されます。dpu_inception_v1_*.xmodel は、VAI_C コンパイラによって生成されたすべての DPU xmodel ファイルを含めるという意味です。

```
aarch64-xilinx-linux-gcc -fPIC -shared \
dpu_inception_v1_*.xmodel -o libdpumodelinception_v1.so
```

ホストの Vitis AI クロス コンパイル環境内では、代わりに次のコマンドを使用します。

```
source /opt/petalinux/2020.2/environment-setup-aarch64-xilinx-linux
CC -fPIC -shared dpu_inception_v1_*.elf -o libdpumodelinception_v1.so
```

注記: DPU の計算能力やコア数は評価ボードごとに異なるため、マルチスレッド Inception-v1 サンプルの最高のスループットを実現するスレッド数は、評価ボードに依存します。各評価ボードの DPU シグネチャ情報は、dexplorer -w を使用して表示します。

miniResNet (Python を使用)

dnndk_sample_base/mini_resnet_py には、Vitis AI の高度な Python API を使用して開発された TensorFlow miniResNet ネットワークの画像分類サンプルが含まれています。コマンド `python3 mini_resnet.py` を実行すると、最も確率の高い結果の上位 5 つが表示されます。miniResNet については、2 番目のブック「Practitioner Bundle of the Deep Learning for Computer Vision with Python」で説明しています。これは元の ResNet-50 モデルをカスタマイズしたもので、同シリーズの第 3 卷「ImageNet Bundle」でも詳しく説明されています。

YOLO-v3 (Python を使用)

dnndk_sample_base/tf_yolov3_voc_py に、Vitis AI の高度な Python API を使用して開発された TensorFlow YOLOv3 ネットワークの画像分類サンプルが含まれています。コマンド `python3 tf_yolov3_voc.py` を実行すると、オブジェクト検出後の結果の画像が表示されます。

エッジ向けの DNNDK プログラミング

DNNDK レガシ プログラミング インターフェイスは、動作時の詳細な DPU 制御を可能にします。これらは、DPU カーネルの読み込み、タスクのインスタンシエーション、および呼び出しのカプセル化を実装することで、DPU タスクのスケジューリング、監視、プロファイリング、およびリソース管理のために XRT または DPU ドライバーを呼び出すことができます。これらの API を使用すると、ザイリンクスの Zynq® UltraScale+™ および Zynq® UltraScale+™ MPSoC デバイスのさまざまなエッジ シナリオの下で、多様な要件に柔軟に対応できます。

プログラミング モデル

DPU プログラミング モデルを理解することで、エッジ DPU を利用する深層学習アプリケーションの開発と運用が容易になります。関連する基本概念には、DPU カーネル、DPU タスク、DPU ノード、DPU テンソルが含まれます。

DPU カーネル

Vitis AI コンパイラによってコンパイルされた後、ニューラル ネットワーク モデルは同等の DPU アセンブリ ファイルに変換され、Deep Neural Network Assembler (DNNAS) によって 1 つの ELF オブジェクト ファイルに組み立てられます。DPU ELF オブジェクト ファイルは DPU カーネルとして見なされ、API `dpuLoadKernel()` を呼び出した後、ランタイム N2Cube にとって 1 つの実行ユニットになります。N2Cube は、DPU 命令とネットワーク パラメーターを含む DPU カーネルを DPU 専用メモリ空間に読み込み、ハードウェア リソースを割り当てます。その後、`dpuCreateTask()` を呼び出して各 DPU カーネルを複数の DPU タスクにインスタンシエートすると、マルチスレッド プログラミングが可能となります。

DPU タスク

各 DPU タスクは、DPU カーネルの実行エンティティです。独自のプライベートメモリ空間があるため、マルチスレッド アプリケーションを使用して複数のタスクを並列処理し、効率性とシステム スループットを向上させることができます。

DPU ノード

DPU ノードは、DPU 上で運用されるネットワーク モデルを構成する基本要素と考えられます。各 DPU ノードは、入力、出力、およびいくつかのパラメーターに関連付けられています。すべての DPU ノードには固有の名前があり、Vitis AI でエクスポートした API がその情報にアクセスできるようにしています。

ノードには、3つのタイプ(境界入力ノード、境界出力ノード、内部ノード)があります。

- 境界入力ノード: 境界入力ノードとは、DPU カーネル トポロジにプリカーサーを持たないノードであり、通常はカーネル内の最初のノードです。1つのカーネルに複数の境界入力ノードが存在することもあります。
- 境界出力ノード: 境界出力ノードとは、DPU カーネル トポロジに後続のノードが存在しないノードです。
- 内部ノード: 境界入力ノードまたは境界出力ノードのいずれでもないノードは、内部ノードと見なされます。

コンパイル後、VAI_C はカーネルとその境界入力/出力ノードに関する情報を提供します。次の図は、Inception-v1 をコンパイルした後の例を示しています。DPU kernel 0 の場合、conv1_7x7_s2 が境界入力ノードで、loss3_classifier が境界出力ノードです。

図 33: VAI_C のコンパイル ログの例

```
[VAI_C][Warning] layer [loss3_loss3] (type: Softmax) is not supported in DPU, deploy it in CPU instead.

Kernel topology "inception_v1_kernel_graph.jpg" for network "inception_v1"
kernel list info for network "inception_v1"
    Kernel ID : Name
        0 : inception_v1_0
        1 : inception_v1_1

    Kernel Name : inception_v1_0
-----
    Kernel Type : DPUKernel
        Code Size : 0.20MB
        Param Size : 6.67MB
        Workload MACs : 3165.34MOPS
        IO Memory Space : 0.76MB
        Mean Value : 104, 117, 123,
    Total Tensor Count : 110
    Boundary Input Tensor(s) (H*W*C)
        data:0(0) : 224*224*3

    Boundary Output Tensor(s) (H*W*C)
        loss3_classifier:0(0) : 1*1*1000

    Total Node Count : 76
        Input Node(s) (H*W*C)
        conv1_7x7_s2(0) : 224*224*3

        Output Node(s) (H*W*C)
        loss3_classifier(0) : 1*1*1000


    Kernel Name : inception_v1_1
-----
    Kernel Type : CPUKernel
    Boundary Input Tensor(s) (H*W*C)
        loss3_loss3:0(0) : 1*1*1000

    Boundary Output Tensor(s) (H*W*C)
        loss3_loss3:0(0) : 1*1*1000

    Input Node(s) (H*W*C)
        loss3_loss3 : 1*1*1000

    Output Node(s) (H*W*C)
        loss3_loss3 : 1*1*1000
```

dpuGetInputTensor*/dpuSetInputTensor* を使用する場合、境界入力ノードを指定するには nodeName パラメーターが必要です。有効な境界入力ノードに対応しない nodeName を使用すると、Vitis AI では次のようなエラーメッセージが表示されます。

```
[DNNDK] Node "xxx" is not a Boundary Input Node for Kernel inception_v1_0.  
[DNNDK] Refer to DNNDK user guide for more info about "Boundary Input Node".
```

同様に、dpuGetOutputTensor*/dpuSetOutputTensor* を使用する場合、有効な境界出力ノードに対応しない「nodeName」を使用すると、次のようなエラーが生成されます。

```
[DNNDK] Node "xxx" is not a Boundary Output Node for Kernel inception_v1_0.  
[DNNDK] Please refer to DNNDK user guide for more info about "Boundary Output Node".
```

DPU テンソル

DPU テンソルは、実行中に情報を保存するために使用する多次元配列のデータ構造です。テンソル プロパティ (高さ、幅、チャネルなど) は、Vitis AI の高度なプログラミング API を使用して取得できます。

標準イメージの場合、イメージ ボリューム用のメモリ レイアウトは、通常、連続バイト ストリームとして CHW (Channel * Height * Width) 形式で保存されます。DPU の場合、入力テンソルおよび出力テンソルのメモリ ストリージ レイアウトは、HWC (Height * Width * Channel) 形式です。DPU テンソル内部のデータは、パディングなしの符号付き 8 ビット整数値の連続ストリームとして保存されます。したがって、DPU 入力テンソルにデータを供給するとき、または DPU 出力テンソルから結果データを取得するときは、このレイアウトの違いに注意する必要があります。

プログラミング インターフェイス

Vitis AI の高度な C++/Python API が導入され、エッジ DPU の深層学習アプリケーション開発が円滑になりました。各 API の詳細は、[付録 A: Vitis AI プログラミング インターフェイス](#) を参照してください。

Python プログラミング API は迅速なネットワーク モデル開発を促進するために提供されており、モデルのトレーニング段階で生成された前処理および後処理の Python コードを再利用します。詳細は、[付録 A: Vitis AI プログラミング インターフェイス](#) を参照してください。DPU 向けに Vitis AI でプログラムする場合、CPU と DPU 間でデータを行き来させることは一般的です。たとえば、CPU で前処理されたデータが DPU に送られて処理されます。その後、DPU で生成された出力は、後処理のために CPU へ送られる必要があります。このような動作に対応するため、Vitis AI は CPU と DPU 間のデータ移動を容易にする一連の API を提供します。それらのいくつかを次に示します。これらの API の使用法は、Caffe や TensorFlow のネットワーク モデルを運用する場合と同じです。

Vitis AI は、計算レイヤーやノードの入力テンソルを設定するために、次の API を提供しています。

- dpuSetInputTensor()
- dpuSetInputTensorInCHWInt8()
- dpuSetInputTensorInCHWFP32()
- dpuSetInputTensorInHWCInt8()
- dpuSetInputTensorInHWCFP32()

Vitis AI は、計算レイヤーやノードから出力テンソルを取得するために、次の API を提供しています。

- dpuGetOutputTensor()
- dpuGetOutputTensorInCHWInt8()
- dpuGetOutputTensorInCHWFP32()

- dpuGetOutputTensorInHWCInt8()
- dpuGetOutputTensorInHWCFP32()

Vitis AI は、DPU 入力テンソルや出力テンソルの開始アドレス、サイズ、量子化係数、および形状情報を取得するために、次の API を提供しています。このような情報をを利用して、ユーザーは自由に前処理のソース コードを実装して符号付きの 8 ビット整数データを DPU に供給したり、後処理のソース コードを実装して DPU 出力データを取得できます。

- dpuGetTensorAddress()
- dpuGetTensorSize()
- dpuGetTensorScale()
- dpuGetTensorHeight()
- dpuGetTensorWidth()
- dpuGetTensorChannel()

Caffe モデル

Caffe フームワークの場合、モデルの前処理が固定されています。Vitis AI は、画像のスケーリング、正規化、量子化など CPU 側で前処理を実行するために、dpuSetInputImage() や dpuSetInputImageWithScale() などの最適化済みのルーチンをいくつか提供しています。その後、データはさらなる処理のために DPU へ送信されます。これらのルーチンは、Vitis AI サンプルのパッケージ内にあります。詳細は、DNNDK サンプル ResNet-50 のソースコードを参照してください。

TensorFlow モデル

TensorFlow フームワークは、入力画像に BGR や RGB 色空間を使用するなど、モデルトレーニング時に非常に柔軟な前処理をサポートしています。したがって、最適化済みルーチン dpuSetInputImage() および dpuSetInputImageWithScale() は、TensorFlow モデルを運用しているときに、直接使用することはできません。前処理コードを実装する必要があります。

次のコード スニペットは、TensorFlow モデルの DPU 入力テンソルに画像を読み込む例を示しています。DPU の入力テンソルに入力される画像の色空間は、モデルトレーニング中に使用される形式と同じにする必要があります。data[j*image.rows*3+k*3+2-i] 場合、画像は RGB 色空間で DPU に供給されます。image.at<Vec3b>(j,k)[i])/255.0 - 0.5)*2 * scale のプロセスは、運用されているモデルに固有であるため、実際に使用するモデルに応じて変更する必要があります。

```
void setInputImage(DPUTask *task, const string& inNode, const cv::Mat& image) {
    DPUTensor* in = dpuGetInputTensor(task, inNode);
    float scale = dpuGetTensorScale(in);
    int width = dpuGetTensorWidth(in);
    int height = dpuGetTensorHeight(in);
    int size = dpuGetTensorSize(in);
    int8_t* data = dpuGetTensorAddress(in);

    for(int i = 0; i < 3; ++i) {
        for(int j = 0; j < image.rows; ++j) {
            for(int k = 0; k < image.cols; ++k) {
                data[j*image.rows*3+k*3+2-i] =

```

```

        }
    }
}
(f1oat(image.at<Vec3b>(j,k)[i])/255.0 - 0.5)*2 * scale;

```

Python は、TensorFlow モデルのトレーニングに使用される非常に一般的な言語です。Vitis AI の高度な Python API を使用した場合、トレーニング段階でこれらの前処理および後処理の Python コードを再利用できます。これは、すばやく評価する目的で DPU 上でモデル運用ワークフローを迅速化するのに役立ちます。その後、C++ コードに変換して、量産要件に対応するためにさらなる性能向上を図ることができます。DNNDK のサンプル miniResNet は、Python を使用して TensorFlow miniResNet モデルを運用するためのリファレンスを提供しています。

DPU のメモリ モデル

エッジ DPU では、Vitis AI コンパイラとランタイム N2Cube が連携して、2 つの異なる DPU メモリ モデル(固有のメモリ モデルとスプリット I/O モデル)をサポートしています。固有のメモリ モデルは、ネットワーク モデルが DPU カーネルにコンパイルされるときのデフォルト モデルです。スプリット I/O モデルを有効にする場合は、ネットワーク モデルをコンパイルするときに、`--split-io-mem` オプションをコンパイラに指定します。

固有のメモリ モデル

このモードの場合、各 DPU タスクに関しては、そのすべての境界入力テンソルと出力テンソル、そして中間機能マップが 1 つの物理的に連続したメモリ バッファー内に収められ、`dpuCreateTask()` を呼び出して 1 つの DPU カーネルから 1 つの DPU タスクをインスタンシエートするときに自動的に割り当てられます。この DPU メモリ バッファーはキャッシュ化して、Arm® CPU 側からのメモリ アクセスを最適化できます。キャッシュのフラッシュと無効化は N2Cube で管理されるため、ユーザーが DPU のメモリ管理やキャッシュ操作に関与する必要はありません。固有のメモリ モデルを使用してモデルを運用することは非常に簡単であり、ほとんどの Vitis™ AI サンプルがこれに該当します。

固有のメモリ モデルの場合、前処理後に入力された Int8 型のデータを DPU タスクのメモリ バッファーの境界入力テンソルにコピーする必要があります。その後、DPU タスクを起動して実行できます。前処理済みの Int8 型入力データが(DPU から直接アクセス可能な)物理的な連続メモリ バッファー内に既にある場合は、追加のオーバーヘッドが生じる可能性があります。1 つの例は、カメラベースの深層学習アプリケーションです。カメラ センサーからの各入力画像に対する前処理のプロセス(画像のスケーリング、モデルの正規化、Float32 から Int8 への量子化など)は、FPGA ロジックを使用して高速化できます。ログ結果データは、物理的な連続メモリ バッファーに記録されます。固有のメモリ モデルを使用する場合は、このデータを DPU 入力メモリ バッファーに再度コピーする必要があります。

スプリット I/O メモリ モデル

スプリット I/O メモリ モデルは、固有のメモリ モデルのリミット問題を解決するために導入され、DPU がほかの物理メモリ バッファーからデータを直接受け取ることができます。`dpuCreateTask()` 関数を呼び出して、`-split-io-mem` オプションでコンパイルされた DPU カーネルから DPU タスクを作成する場合、N2Cube は中間機能マップ用に DPU メモリ バッファーのみを割り当てます。境界入力テンソルと出力テンソル用にそれぞれ物理的な連続メモリ バッファーを割り当てるかどうかは、ユーザーが判断します。入力メモリ バッファーと出力メモリ バッファーのサイズは、コンパイラ ビルド ログのフィールド名(`Input Mem Size` および `Output Mem Size`)で確認できます。これらのメモリ バッファーをキャッシュできる場合、キャッシュ コヒーレンシにも注意する必要があります。

DNNDK のサンプル `split_io` では、スプリット I/O メモリ モデルのプログラミング リファレンスを提供しています。TensorFlow SSD がリファレンスとして使用されます。SSD モデル用には、入力テンソル イメージが 1 つ(`image:0`)、出力テンソル イメージが 2 つ(`ssd300_concat:0`, `ssd300_concat_1:0`)あります。コンパイラのビルド ログから、DPU 入力メモリ バッファー(テンソル イメージ `image:0`)のサイズは 270000 であり、DPU 出力メモリ バッファー(出力テンソル `ssd300_concat:0` および `ssd300_concat_1:0`)のサイズは 218304 であることが

わかります。`dpuAllocMem()` を使用して、これらにメモリ バッファーを割り当てます。その後、実行を開始する前に、`dpuBindInputTensorBaseAddress()` と `dpuBindOutputTensorBaseAddress()` を使用して、入出力メモリ バッファー アドレスを DPU タスクに関連付けます。入力データが DPU 入力メモリ バッファーに入力されると、`dpuSyncMemToDev()` が呼び出されてキャッシュ ラインがフラッシュされます。DPU タスクの実行が完了すると、`dpuSyncDevToMem()` が呼び出されてキャッシュ ラインが無効になります。

注記: 4つの API (`dpuAllocMem()`、`dpuFreeMem()`、`dpuSyncMemToDev()`、`dpuSyncDevToMem()`) は、スプリット IO メモリ モデルのデモ専用に提供されています。量産環境でそのまま使用することを目的としていません。カスタム要件に対応する目的で、これらの機能を実装することは可能です。

DPU コアのアフィニティ

エッジ DPU のランタイム N2Cube は、API `dpuSetTaskAffinity()` を使用して DPU コアのアフィニティをサポートします。これは、DPU タスクを目的の DPU コアに動的に割り当てる場合に使用できるため、必要に応じて DPU コアの割り当てやスケジューリングを操作できます。DPU コアのアフィニティは、`dpuSetTaskAffinity()` の 2 番目の引数 `coreMask` で指定されます。`coreMask` の各ビットが 1 つの DPU コアを表します(最下位ビットはコア 0、2 番目に最下位ビットはコア 1 ... と続く)。一度に複数のマスク ビットを指定できますが、有効な最大 DPU コア数を超えることはできません。たとえば、マスク値 `0x3` は、タスクを DPU コア 0 および 1 に割り当てる能够性を示し、コア 0 または 1 のいずれかが有効であれば、すぐにスケジューリングされます。

優先度ベースの DPU スケジューリング

N2Cube は、API `dpuSetTaskPriority()` を使用して、実行時に DPU タスクの優先度を専用の値に指定することで、優先度ベースの DPU タスクのスケジューリングを可能にします。優先度の範囲は 0(最も高い優先度) ~ 15(最も低い優先度) です。優先度を指定しない場合、DPU タスクの優先度はデフォルトの 15 に設定されます。この機能により、さまざまなエッジ シナリオで生じる多様な要件に対応する柔軟性がもたらされます。同時に実行されるモデルに対して異なる優先度を指定することで、すべて準備が整った状態にある場合に異なる順序で DPU コアにスケジューリングできます。アフィニティが指定されている場合、N2Cube の優先度ベースのスケジューリングも DPU コアのアフィニティに準拠します。

DNNDK の姿勢検出サンプルでは、DPU 優先スケジューリングの機能を示しています。このサンプルでは、2つのモデル(人物検出用の SSD モデル、体のキー ポイント検出用のポーズ検出モデル)が使用されています。SSD モデルは、DPU カーネル `ssd_person` にコンパイルされ、ポーズ検出モデルは、2つの DPU カーネル (`pose_0` および `pose_2`) にコンパイルされます。したがって、各入力イメージは、これら 3 つの DPU カーネルを `ssd_person`、`pose_0`、および `pose_2` の順に実行する必要があります。マルチスレッドの場合には、これら 3 つのカーネルで複数の入力イメージがオーバーラップする可能性があります。レイテンシを改善するために、`ssd_person`、`pose_0`、および `pose_2` の DPU タスクには、それぞれ優先度 3、2、1 が割り当てられ、後者の DPU カーネルの DPU タスクが高い優先度でスケジューリングされます。

DNNDK ユーティリティ

- DSight: DSight は、エッジ DPU 向けの Vitis AI 性能プロファイラーであり、モデルの性能をプロファイリングするための視覚分析ツールです。次の図に、使用方法を示します。

図 34: DSight のヘルプ情報

```
root@xlnx:~# dsight -h
Usage: dsight <option>
Options are:
  -p  --profile      Specify DPU trace file for profiling
  -v  --version       Display DSight version info
  -h  --help          Display this information
```

ランタイム N2cube で生成されたログ ファイルを処理することにより、DSight は HTML ウェブ ページを生成し、DPU コアの使用率とスケジューリング効率を示す視覚的なチャートを提供できます。

- DExplorer: DExplorer は、ターゲット ボード上で動作するユーティリティです。DPU 動作モードの設定、DNNDK バージョンの確認、DPU ステータスの確認、および DPU コア シグネチャの確認の機能があります。次の図は、DExplorer の使用に関するヘルプ情報を示しています。

図 35: DExplorer の使用オプション

```
Usage: dexplorer <option>
Options are:
  -v  --version      Display version info for each DNNDK component
  -s  --status       Display the status of DPU cores
  -w  --whoami        Display the info of DPU cores
  -m  --mode         Specify DNNDK N2Cube running mode: normal, profile, or debug
  -t  --timeout      Specify DPU timeout limitation in seconds under integer range of [1,
  -h  --help          Display this information
```

- DNNDK のバージョンを確認: `dexplore -v` を実行すると、DNNDK の各コンポーネント (N2Cube、DPU ドライバー、DExplorer、および DSight) のバージョン情報が表示されます。
- DPU のステータスを確認: N2cube の実行モード、DPU タイムアウトしきい値、DPU デバッグ レベル、DPU コアのステータス、DPU レジスタ情報、DPU メモリリソース、使用率などの DPU のステータス情報を提供します。次の図に、DPU ステータスのスクリーンショットを示します。

図 36: DExplorer のステータス

```
root@dp-n1:~# dexplorer -s
[DPU cache]
Enabled

[DPU mode]
normal

[DPU timeout limitation (in seconds)]
5

[DPU Debug Info]
Debug level      : 9
Core 0 schedule : 0
Core 0 interrupt: 0

[DPU Resource]
DPU Core        : 0
State            : Idle
PID              : 0
TaskID           : 0
Start            : 0
End              : 0

[DPU Registers]
VER              : 0x05c1c6bd
RST              : 0x000000ff
ISR              : 0x00000000
IMR              : 0x00000000
IRSR             : 0x00000000
ICR              : 0x00000000

DPU Core        : 0
HP_CTL           : 0x07070f0f
ADDR_IO          : 0x00000000
ADDR_WEIGHT      : 0x00000000
ADDR_CODE         : 0x00000000
ADDR_PROF        : 0x00000000
```

- DPU 実行モードの設定: エッジ DPU のランタイム N2cube は、Vitis AI アプリケーションのデバッグおよびプロファイリングに役立つ 3 種類の DPU 実行モードをサポートしています。
 - 通常モード: 通常モードではオーバーヘッドが発生しないため、DPU アプリケーションを最大の性能で実行できます。
 - プロファイル モード: プロファイル モードの場合、DPU のプロファイリング スイッチがオンになります。プロファイル モードで深層学習アプリケーションを実行すると、N2Cube はニューラルネットワーク実行時にレイヤーごとの性能データをコンソールに出力します。これと同時に、`dpu_trace_[PID].prof` という名前のプロファイルが、カレント フォルダーに生成されます。このファイルは、DSight ツールで使用できます。
 - デバッグ モード: このモードの場合、DPU は、実行中に各 DPU 計算ノードのロー データ (バイナリ形式の DPU 命令コード、ネットワーク パラメーター、DPU 入力テンソル、出力テンソル) をダンプします。これにより、DPU アプリケーションをデバッグして問題を特定できます。

注記: Profile モードと Debug モードは、Vitis AI コンパイラによってデバッグ モードの DPU ELF オブジェクトにコンパイルされたネットワーク モデルに対してのみ有効です。

- DPU のシグネチャを確認: ザイリンクスの各種 FPGA デバイスでさまざまな深層学習アクセラレーションの要件を満たすために、新しい DPU コアが導入されています。たとえば、DPU アーキテクチャとして B1024F、B1152F、B1600F、B2304F、B4096F を利用できます。深層学習アルゴリズムは急速に改良が進んでいるため、これらをサポートするために、各 DPU アーキテクチャはさまざまなバージョンの DPU 命令セット (DPU ターゲットバージョン) を実装しています。DPU シグネチャとは、特定の DPU アーキテクチャ バージョンの仕様情報を指し、これにはターゲットバージョン、動作周波数、DPU コア数、ハード アクセラレーション モジュール (Softmax など) といった情報が含まれます。-w オプションを指定すると、DPU シグネチャを確認できます。次の図に、DExplorer -w を実行した場合の画面の例を示します。設定可能な DPU の場合、次の図に示すように、DExplorer は DPU シグネチャのすべてのコンフィギュレーション パラメーターを表示するのに役立ちます。

図 37: DPU シグネチャの設定パラメーターの例

```
root@xilinx-zcu102-2019_1:~$dexplorer -w
[DPU IP Spec]
IP Timestamp      : 2019-07-24 11:15:00
DPUs Core Count   : 3

[DPU Core Configuration List]
DPUs Core          : #0
DPUs Enabled       : Yes
DPUs Arch          : B4096
DPUs Target Version : v1.4.0
DPUs Frequency     : 325 MHz
Ram Usage          : Low
DepthwiseConv      : Enabled
DepthwiseConv+Relu6 : Enabled
Conv+Leakyrelu    : Enabled
Conv+Relu6         : Enabled
Channel Augmentation : Enabled
Average Pool       : Enabled
DPUs Core          : #1
```

- DDump: DDump は、DPU ELF ファイル、ハイブリッド実行可能ファイル、または DPU 共有ライブラリ内にカプセル化された情報をダンプするユーティリティ ツールであり、さまざまな問題を分析してデバッグするのに役立ちます。詳細は、「DPU 共有ライブラリ」を参照してください。DDump は、ランタイム コンテナー vitis-ai-docker-runtime と Vitis AI 評価ボードの両方で利用できます。次の図に使用法を示します。ランタイム コンテナーの場合、パス /opt/vitis-ai/utility/ddump からアクセスできます。評価ボードの場合は、Linux システム パスの下にインストールされるため、直接使用できます。

図 38: DDump の使用オプション

```
DDump - Xilinx DNNDK utility to parse and dump DPU ELF file or
       DPU hybrid executable file
Usage: ddump <option>
At least one of the following switches must be given:
  -f  --file      Specify DPU hybrid executable or DPU ELF object
  -k  --klist     Display each kernel general info from DPU ELF file
                 or DPU hybrid executable file
  -d  --dpu       Display DPU architecture info for each kernel
  -c  --compiler  Display the DNCC compiler version for each kernel
  -a  --all        Display all above info
  -v  --version   Display DDump version info
  -h  --help       Display this help info
```

- DPU カーネル情報を確認する: DDump は、DPU ELF ファイル、ハイブリッド実行可能ファイル、または DPU 共有ライブラリから DPU カーネルごとに次の情報をダンプできます。
 - Mode: VAI_C コンパイラでコンパイルされた DPU カーネルのモード (NORMAL、DEBUG)。
 - Code Size: DPU カーネルの DPU 命令コード サイズ (単位: MB、KB、バイト)。
 - Param Size: DPU カーネルのパラメーター サイズ (重みとバイアスを含む) (単位: MB、KB、バイト)。
 - Workload MACs: DPU カーネルの計算ワークロード (単位: MOPS)。

- I/O Memory Space: 中間機能マップに必要な DPU メモリ空間(単位: MB、KB、バイト)。作成された DPU タスクごとに、N2Cube が中間機能マップ用の DPU メモリ バッファーを自動的に割り当てます。
- Mean Value: DPU カーネルの平均値。
- Node Count: DPU カーネルの DPU ノードの総数。
- Tensor Count: DPU カーネルの DPU テンソルの総数。
- Tensor In(H*W*C): DPU 入力テンソルのリストとそれらの形状情報。HWC(高さ*幅*チャネル)形式。
- Tensor Out(H*W*C): DPU 出力テンソルのリストとそれらの形状情報。HWC(高さ*幅*チャネル)形式。

次の図に、ddump -f dpu_resnet50_0.elf -k コマンドで出力された ResNet-50 DPU ELF ファイル dpu_resnet50_0.elf の DPU カーネル情報を示します。

図 39: DDump を使用した ResNet50 の DPU カーネル情報

```
DPU Kernel List from file dnnc_output/dpu_resnet50_0.elf
ID: Name
0: resnet50_0

DPU Kernel name: resnet50_0
-----
-> DPU Kernel general info
    Mode: NORMAL
    Code Size: 1.28MB
    Param Size: 24.35MB
    Workload MACs: 7358.50MOPS
    IO Memory Space: 2.25MB
    Mean Value: 104, 107, 123
    Node Count: 55
    Tensor Count: 56
    Tensor In(H*W*C)
    Tensor ID-0: 224*224*3
    Tensor Out(H*W*C)
    Tensor ID-55: 1*1*1000
```

- DPU アーキテクチャ情報を確認する: DPU DCF からの DPU コンフィギュレーション情報は、各 DPU カーネルの VAI_C コンパイラによって DPU ELF ファイルに自動的にラップされます。その後、VAI_C は DPU コンフィギュレーション パラメーターに従って適切な DPU 命令を生成します。コンフィギュレーション可能な DPU 記述の詳細は、『Zynq DPU v3.1 IP 製品ガイド』(PG338: 英語版、日本語版) を参照してください。DDump は、次に示す DPU アーキテクチャ情報をダンプできます。
 - DPU Target Ver: DPU 命令セットのバージョン。
 - DPU Arch Type: DPU アーキテクチャの種類(B512、B800、B1024、B1152、B1600、B2304、B3136、B4096など)。
 - RAM の使用: RAM 使用量 (low または high)。
 - DepthwiseConv: DepthwiseConv エンジンの有効/無効。
 - DepthwiseConv+Relu6: Relu6 に続く DepthwiseConv の演算子パターンの有効/無効。
 - Conv+Leakyrelu: Leakyrelu に続く Conv の演算子パターンの有効/無効。
 - Conv+Relu6: Relu6 に続く Conv の演算子パターンの有効/無効。

- Channel Augmentation: DPU のチャネル並列度をはるかに下回る数の入力チャネルを処理するレイヤーで、DPU の効率を向上させるために使用するオプション機能。
- Average Pool: 平均プーリング エンジンの有効/無効。

DPU アーキテクチャ情報は、DPU IP のバージョンによって異なる場合があります。ddump -f dpu_resnet50_0.elf -d コマンドを実行すると、次の図に示すように、VAI_C が ResNet-50 モデルをコンパイルする際に使用する DPU アーキテクチャ情報が出力されます。

図 40: ResNet50 の DDump DPU アーキテクチャ情報

```
DPU Kernel List from file dpu_resnet50_0.elf
      ID: Name
      0: resnet50_0

DPU Kernel name: resnet50_0
-----
-> DPU architecture info
      DPU ABI Ver: v2.0
DPU Configuration Parameters
      DPU Target Ver: 1.4.0
      DPU Arch Type: B512
      RAM Usage: high
      DepthwiseConv: Enabled
      DepthwiseConv+Relu6: Enabled
      Conv+Leakyrelu: Enabled
      Conv+Relu6: Enabled
      Channel Augmentation: Enabled
      Average Pool: Disabled
```

- VAI_C 情報を確認する: VAI_C バージョン情報は、ネットワーク モデルのコンパイル中に自動的に DPU ELF ファイルへ埋め込まれます。DDump は、この VAI_C バージョン情報をダンプするのに役立ちます。この情報は、ユーザーがデバッグ目的で Vitis AI サポート チームに提供できます。コマンド ddump -f dpu_resnet50_0.elf -c を実行すると、次の図に示す ResNet-50 モデルの VAI_C 情報情報が出力されます。

図 41: ResNet50 の DDump VAI_C 情報

```
DPU Kernel List from file dnnc_output/dpu_resnet50_0.elf
      ID: Name
      0: resnet50_0

DPU Kernel name: resnet50_0
-----
-> DNNC compiler info
      DNNC Ver: dnnc version v3.00
DPU Target : v1.4.0
Build Label: Jul 22 2019 16:47:08
Copyright @2019 Xilinx Inc. All Rights Reserved.
```

- Legacy Support: DDump は、生成されたレガシ DPU ELF ファイル、ハイブリッド実行ファイル、および DPU 共有ライブラリの情報をダンプすることも可能です。この場合の主な違いは、詳しい DPU アーキテクチャ情報がないことです。次の図に、コマンド `ddump -f dpu_resnet50_0.elf -a` を使用して、レガシ ResNet-50 DPU ELF ファイルのすべての情報をダンプする例を示します。
- DLet: DLet は、Vivado® Design Suite で生成された DPU ハードウェア ハンドオフ ファイル (HWH) からエッジ DPU のさまざまなコンフィギュレーション パラメーターを解析/抽出することを目的とするホスト ツールです。次の図は、DLet の使用オプション情報を示しています。

図 42: DLet の使用オプション

```

Usage: dlet <option>
Options are:
  -v  --version      Display version of DLet
  -f  --file        Specity hardware hand-off(HWH) file
  -h  --help         Display the usage of DLet
    
```

Vivado プロジェクトの場合、DPU HWH はデフォルトで次のディレクトリにあります。`<prj_name>` は、Vivado のプロジェクト名で、`<bd_name>` は Vivado のブロック デザイン名です。

```

<prj_name>/<prj_name>.srcs/sources_1/bd/<bd_name>/hw_handoff/
<bd_name>.hwh
    
```

コマンド `dlet -f <bd_name>.hwh` を実行すると、DLet は `dpu-dd-mm-yyyy-hh-mm.dcf`. `dd-mm-yyyy-hh-mm` の形式で名前が付けられた DCF (DPU コンフィギュレーション ファイル) を出力します。dd-mm-yyyy-hh-mm は、DPU HWH が作成されたときのタイム スタンプです。指定された DCF ファイルを使用して、VAI_C コンパイラは DPU コンフィギュレーション パラメーターに適した DPU コード命令を自動生成します。

DNNDK プロファイルを使用したプロファイリング

DSight は DNNDK のパフォーマンス プロファイリング ツールです。ニューラル ネットワーク モデルのプロファイリング結果を視覚的に解析できます。次の図に、使用方法を示します。

図 43: DSight のヘルプ情報

```

root@xlnx:~# dsight -h
Usage: dsight <option>
Options are:
  -p  --profile      Specify DPU trace file for profiling
  -v  --version      Display DSight version info
  -h  --help         Display this information
    
```

DSight は、N2Cube のトレーサーによって生成されたログ ファイルを処理して HTML ファイルを生成します。このファイルを使用して、ニューラル ネットワーク モデルを視覚的に解析できます。プロファイルの使用手順は、次のとおりです。

1. コマンド `dexplorer -m profile` を実行し、N2Cube をプロファイル モードに設定します。
2. 深層学習アプリケーションを実行します。実行が完了すると、`dpu_trace_[PID].prof` という名前のプロファイル ファイルが生成され、さらにチェックと解析ができます (PID は深層学習アプリケーションのプロセス ID)。

3. コマンド `dsight -p dpu_trace_[PID].prof` を実行して、DSight ツールで HTML ファイルを生成します。`dpu_trace_[PID].html` という名前の HTML ファイルが生成されます。
4. 生成された HTML ファイルをウェブ ブラウザーで開きます。

細粒度のプロファイリング

モデルがエッジ DPU にコンパイルされて展開された後、ユーティリティの DExplorer を使用して細粒度のプロファイリングを実行し、レイヤーごとの実行時間と DDR メモリ帯域幅を確認できます。これは、モデルの性能ボトルネックを分析するのに非常に役立ちます。

注記: モデルは、Vitis AI コンパイラを使用してデバッグ モードのカーネルにコンパイルする必要があります。細粒度プロファイリングは、通常モードのカーネルに使用できません。

デバッグ モードのカーネルの細粒度プロファイリングを有効にするには、2 つのアプローチがあります。

- DPU アプリケーションの実行を開始する前に、`dexplorer -m profile` を実行します。これにより、N2Cube グローバル実行モードが変更され、すべての DPU タスク(デバッグ モード)がプロファイリング モードで実行されます。
- 専用の DPU タスクのみにプロファイリング モードを有効にするには、`dpuCreateTask()` を使用して、`flag_T_MODE_PROF` または `dpuEnableTaskProfile()` を指定します。その他のタスクには影響を及ぼしません。

次の図に、ResNet50 モデルのプロファイリングのスクリーン キャプチャを示しています。ResNet-50 カーネル上の各 DPU レイヤー(またはノード)のプロファイリング情報が一覧表示されています。

注記: Vitis AI コンパイラは複数のレイヤー/演算子の融合を実行して実行性能と DDR メモリ アクセスを最適化するため、各 DPU ノードには、オリジナル Caffe または TensorFlow モデルのレイヤーまたは演算子が含まれる場合があります。

図 44: ResNet50 の細粒度プロファイリング

ID	NodeName	Workload(MOP)	Mem(MB)	RunTime(ms)	Perf(GOPS)	Utilization	MB/S
0	conv1	236.0	0.4	5.28	44.7	19.4%	67
1	res2a_branch2a	25.7	0.4	0.23	113.2	49.2%	1719
2	res2a_branch1	102.8	1.0	0.95	108.5	47.2%	1044
3	res2a_branch2b	231.2	0.4	1.34	172.0	74.8%	314
4	res2a_branch2c	102.8	1.0	1.62	63.3	27.5%	616
5	res2b_branch2a	102.8	1.0	0.65	159.3	69.3%	1518
6	res2b_branch2b	231.2	0.4	1.34	172.0	74.8%	314
7	res2b_branch2c	102.8	1.0	1.62	63.6	27.6%	619
8	res2c_branch2a	102.8	1.0	0.64	159.8	69.5%	1523
9	res2c_branch2b	231.2	0.4	1.35	171.9	74.7%	313
10	res2c_branch2c	102.8	1.0	1.62	63.3	27.5%	616
11	res3a_branch2a	51.4	0.9	0.41	125.3	54.5%	2197
12	res3a_branch1	295.5	1.3	1.49	137.8	59.9%	870
13	res3a_branch2b	231.2	0.3	1.14	202.6	88.1%	294
14	res3a_branch2c	102.8	0.6	1.22	84.6	36.8%	466
15	res3b_branch2a	102.8	0.5	0.58	176.6	76.8%	939
16	res3b_branch2b	231.2	0.3	1.14	202.6	88.1%	294
17	res3b_branch2c	102.8	0.6	1.21	84.6	36.8%	466
18	res3c_branch2a	102.8	0.5	0.58	176.0	76.5%	936
19	res3c_branch2b	231.2	0.3	1.14	202.6	88.1%	294
20	res3c_branch2c	102.8	0.6	1.21	84.6	36.8%	466
21	res3d_branch2a	102.8	0.5	0.58	176.0	76.5%	936
22	res3d_branch2b	231.2	0.3	1.14	202.5	88.0%	294
23	res3d_branch2c	102.8	0.6	1.21	84.9	36.9%	468
24	res4a_branch2a	51.4	0.6	0.49	105.9	46.1%	1164
25	res4a_branch1	295.5	1.1	2.01	102.0	44.4%	551
26	res4a_branch2b	231.2	0.7	1.34	172.9	75.2%	497
27	res4a_branch2c	102.8	0.5	1.01	101.8	44.3%	508
28	res4b_branch2a	102.8	0.5	0.71	145.1	63.1%	700
29	res4b_branch2b	231.2	0.7	1.34	172.9	75.2%	497
30	res4b_branch2c	102.8	0.5	1.00	102.9	44.7%	513
31	res4c_branch2a	102.8	0.5	0.71	144.5	62.8%	697
32	res4c_branch2b	231.2	0.7	1.34	172.7	75.1%	496
33	res4c_branch2c	102.8	0.5	1.01	101.7	44.2%	508
34	res4d_branch2a	102.8	0.5	0.71	145.3	63.2%	701
35	res4d_branch2b	231.2	0.7	1.34	172.3	74.9%	495
36	res4d_branch2c	102.8	0.5	1.02	100.9	43.9%	504
37	res4e_branch2a	102.8	0.5	0.71	145.3	63.2%	701
38	res4e_branch2b	231.2	0.7	1.34	172.0	75.1%	496
39	res4e_branch2c	102.8	0.5	1.01	101.8	44.3%	508
40	res4f_branch2a	102.8	0.5	0.70	146.6	63.7%	707
41	res4f_branch2b	231.2	0.7	1.34	172.8	75.1%	496
42	res4f_branch2c	102.8	0.5	1.01	101.5	44.1%	507
43	res5a_branch2a	51.4	0.7	0.70	73.6	32.0%	1044
44	res5a_branch1	295.5	2.3	2.81	73.3	31.9%	835
45	res5a_branch2b	231.2	2.3	1.77	130.6	56.8%	1304
46	res5a_branch2c	102.8	1.2	1.32	77.8	33.8%	875
47	res5b_branch2a	102.8	1.1	1.01	101.8	44.3%	1120
48	res5b_branch2b	231.2	2.3	1.77	130.7	56.8%	1304
49	res5b_branch2c	102.8	1.2	1.33	77.3	33.6%	869
50	res5c_branch2a	102.8	1.1	1.01	101.9	44.3%	1121
51	res5c_branch2b	231.2	2.3	1.78	130.0	56.5%	1298
52	res5c_branch2c	102.8	1.2	1.28	80.2	34.9%	900
Total Nodes In Avg:							
All	7711.9	44.4	64.62	119.3	51.9%	687	

次のフィールドが含まれます。

- ID: DPU ノードのインデックス ID。
- NodeName: DPU ノード名。
- Workload (MOP): 計算ワークロード (MAC では 2 つの演算子を示す)。
- Mem (MB): この DPU ノードのコード、パラメーター、機能マップのメモリ容量。
- Runtime (ms): 実行時間 (ms)。
- Perf (GOPS): DPU の性能 (GOPS)。
- Utilization (%): DPU の使用率 (%)。
- MB/S: 平均の DDR メモリ アクセス帯域幅。

DSight による 1 つのモデルの細粒度プロファイリングの結果、DPU コアの性能が不十分な場合は、性能が向上するように DPU の構成を変更できます。たとえば、B1152 から B4096 に DPU アーキテクチャを変更したり、大容量のオンチップ RAM に変更するなどの方法があります。詳細は、<https://github.com/Xilinx/Vitis-AI/tree/master/dsa/DPU-TRD> を参照してください。一方、現状の DPU コアで十分な性能が得られる場合は、DPU の構成を変更して、ロジックリソースの要件を緩和することができます。

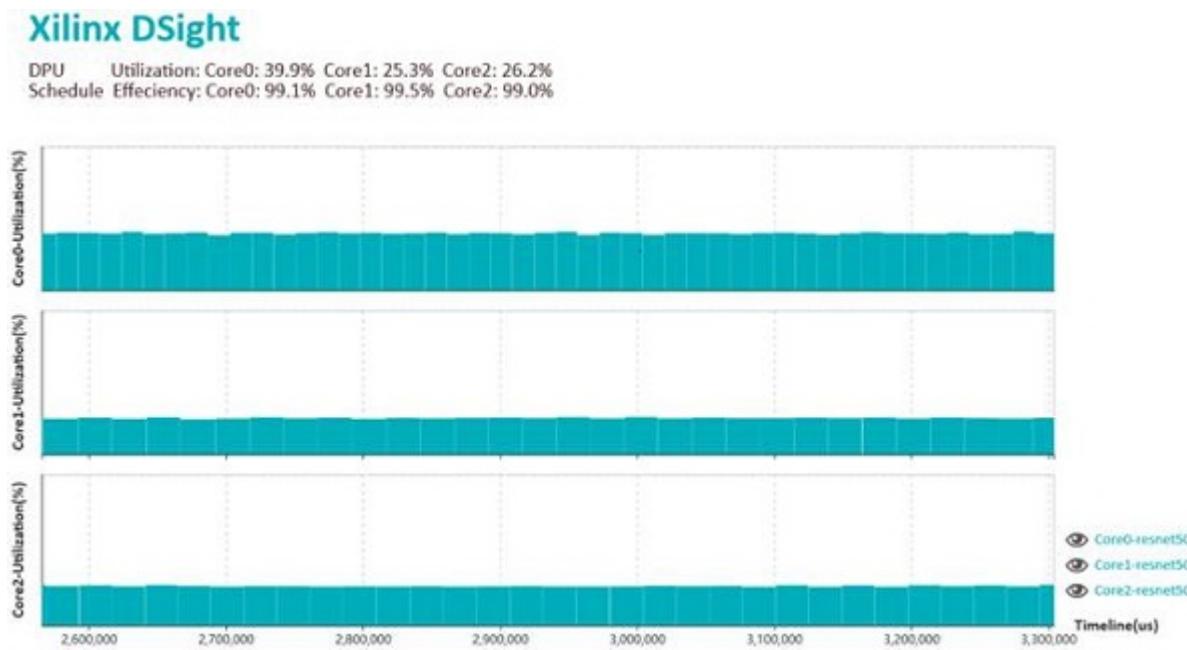
パノラマビュー プロファイリング

DSight は、アプリケーションのボトルネックを見つけて性能をさらに最適化できるように、DPU コアの使用率をパノラマ表示できる視覚形式の統計プロファイルを提供します。パノラマビュー プロファイリングを実行する前に、VAI_C を使用してモデルを通常モードの DPU カーネルにコンパイルすることを推奨しています。

次の手順は、DSight ツールでプロファイリングを実行する方法を示しています。

- コマンド `dexplorer -m` プロファイルを使用して、N2Cube をプロファイル モードに切り替えます。
- DPU アプリケーションを実行し、通常の性能で動作を数秒間続けた後にプロセスを停止します。さらなる処理用として、アプリケーションのディレクトリ内に `dpu_trace_[PID].prof` という名前のプロファイル ファイルが生成されます(PID は、起動した DPU アプリケーションのプロセス ID)。
- コマンド `dsight -p dpu_trace_[PID].prof` を使用して Dsight ツールを起動します。
`dpu_trace_[PID].html` という名前の HTML ファイルが Dsight で生成されます。
- この生成された HTML ウェブ ページを任意のウェブ ブラウザーで開くと、視覚的なチャートが表示されます。次の図に、3 つの DPU コアを対象とするマルチスレッド ResNet-50 のプロファイリング例を示します。

図 45: プロファイリング例



- DPU Utilization (Y-axis): 各 DPU コアの使用率を一覧表示します。パーセンテージが高いほど、DPU の計算能力がフルに活用されてモデルの実行を高速化していることを示します。このパーセンテージを低くするには、DPU コンフィギュレーションを変更して必要なロジックリソースを削減するか、アルゴリズム要件に応じた DPU 計算リソースになるようにアルゴリズムを再設計してください。

- Schedule Efficiency (X-axis): ランタイム N2Cube でスケジューリングされている各コアの割合を示します。パーセンテージが低い場合、DPU コアがトリガーされるチャンスを増やすように、アプリケーションのスレッド数を改善することができます。DPU コアのスケジュール効率をさらに向上させるには、NEON Intrinsic、アセンブリ命令、Vitis アクセラレーション ライブラリ (xfOpenCV など) の使用など、Arm CPU 側で実行されるその他の計算ワークロードを最適化する必要があります。通常、このような DPU 以外のワークロードには、前処理、後処理、または DPU でサポートされていない深層学習演算子などがあります。

DNNDK プログラミング API

このセクションでは、エッジ DPU にのみ使用可能なレガシ DNNDK プログラミング インターフェイスについて説明します。

注記: これらの API は、今後のリリースで廃止されるため、新規アプリケーションやプロジェクトには、VART API を使用してください。

DNNDK レガシ API は、Vitis AI の高度な低レベル C++/Python プログラミング インターフェイスと見なされます。Vitis AI 開発キットは、さまざまなエッジシナリオの下で多様な要件に柔軟に対応できる包括的な API セットを提供します。たとえば、低レベル API `dpuSetTaskPriority()` を使用して、DPU タスクのスケジューリングの優先順位を指定し、指定した優先順位でさまざまなモデルをスケジューリングできます。`dpuSetTaskAffinity()` を使用して、DPU タスクを目的の DPU コアに動的に割り当てることができるため、必要に応じて DPU コアの割り当てやスケジューリングを操作できます。また、このような高度な API は前方互換性を備えているため、既存のソースコードを変更せずに DNNDK のレガシ プロジェクトを Vitis プラットフォームに移植できます。

エッジ DPU の場合、ランタイム ライブラリ `libn2cube` 内に Vitis AI の高度な低レベル C++ API が実装されており、ヘッダーファイル `n2cube.h` 内にヘッダーファイル `dnndk.h` としてエクスポートされます。したがって、ユーザーはソース コードで `dnndk.h` を含めるだけで完了します。

一方、モジュール `n2cube` で適切な低レベルの Python API を適用できます。これは、ライブラリ `libn2cube` 内の C++ API と同等のラッパーです。Python プログラミング インターフェイスを使用すると、モデル トレーニング中に開発した Python コードを再利用して、エッジ DPU にすばやくモデルを展開して評価できます。

C++ API

ここでは、Vitis AI の高度な低レベル C++ プログラミング API について簡単に説明します。

名称

`libn2cube.so`

説明

DPU ランタイム ライブラリ

ルーチン

- `dpuOpen()`: DPU デバイスを開いて使用法を初期化する。
- `dpuClose()`: DPU デバイスの使用法を確定して終了する。
- `dpuLoadKernel()`: DPU カーネルをロードして、コード/重み/バイアス セグメント用に DPU メモリ空間を割り当てる。

- `dpuDestroyKernel()`: DPU カーネルを無効にして、関連するリソースを解放する。
- `dpuCreateTask()`: ある DPU カーネルから 1 つの DPU タスクをインスタンシエートし、そのプライベート ワーキング メモリ バッファーを割り当て、実行コンテキストの準備をする。
- `dpuRunTask()`: DPU タスクの実行を開始する。
- `dpuDestroyTask()`: DPU タスクを削除し、そのワーキング メモリ バッファーを解放して、関連する実行コンテキストを無効にする。
- `dpuSetTaskPriority()`: 実行時に DPU タスクの優先度を指定した値に動的に設定する。優先度の範囲は 0 (最も高い優先度) ~ 15 (最も低い優先度)。指定しない場合、DPU タスクの優先度はデフォルトの 15 に設定される。
- `dpuGetTaskPriority()`: DPU タスクの優先度を取得する。
- `dpuSetTaskAffinity()`: 実行時に DPU コアにおける DPU タスクのアフィニティを動的に設定する。指定しない場合、デフォルトで DPU タスクは有効なすべての DPU コアで実行可能。
- `dpuGetTaskAffinity()`: DPU コアにおける DPU タスクのアフィニティを取得する。
- `dpuEnableTaskDebug()`: デバッグ目的で実行中に DPU タスクのダンプ機能を有効にする。
- `dpuEnableTaskProfile()`: 実行中に DPU タスクのプロファイリング機能を有効にして、性能メトリクスを取得する。
- `dpuGetTaskProfile()`: DPU タスクの実行時間を取得する。
- `dpuGetNodeProfile()`: DPU ノードの実行時間を取得する。
- `dpuGetInputTensorCnt()`: 1 つの DPU タスクの入力テンソルの総数を取得する。
- `dpuGetInputTensor()`: 1 つの DPU タスクの入力テンソルの取得する。
- `dpuGetInputTensorAddress()`: 1 つの DPU タスクの入力テンソルの開始アドレスを取得する。
- `dpuGetInputTensorSize()`: 1 つの DPU タスクの入力テンソルのサイズ (バイト) を取得する。
- `dpuGetInputTensorScale()`: 1 つの DPU タスクの入力テンソルのスケール値を取得する。
- `dpuGetInputTensorHeight()`: 1 つの DPU タスクの入力テンソルの高さを取得する。
- `dpuGetInputTensorWidth()`: 1 つの DPU タスクの入力テンソルの幅を取得する。
- `dpuGetInputTensorChannel()`: 1 つの DPU タスクの入力テンソルのチャネル サイズを取得する。
- `dpuGetOutputTensorCnt()`: 1 つの DPU タスクの出力テンソルの総数を取得する。
- `dpuGetOutputTensor()`: 1 つの DPU タスクの出力テンソルを取得する。
- `dpuGetOutputTensorAddress()`: 1 つの DPU タスクの出力テンソルの開始アドレスを取得する。
- `dpuGetOutputTensorSize()`: 1 つの DPU タスクの出力テンソルのサイズ バイトを取得する。
- `dpuGetOutputTensorScale()`: 1 つの DPU タスクの出力テンソルのスケール値を取得する。
- `dpuGetOutputTensorHeight()`: 1 つの DPU タスクの出力テンソルの高さを取得する。
- `dpuGetOutputTensorWidth()`: 1 つの DPU タスクの出力テンソルの幅を取得する。
- `dpuGetOutputTensorChannel()`: 1 つの DPU タスクの出力テンソルのチャネル サイズを取得する。
- `dpuGetTensorSize()`: 1 つの DPU テンソルのサイズを取得する。

- `dpuGetTensorAddress()`: 1 つの DPU テンソルの開始アドレスを取得する。
- `dpuGetTensorScale()`: 1 つの DPU テンソルのスケール値を取得する。
- `dpuGetTensorHeight()`: 1 つの DPU テンソルの高さを取得する。
- `dpuGetTensorWidth()`: 1 つの DPU テンソルの幅を取得する。
- `dpuGetTensorChannel()`: 1 つの DPU テンソルのチャネル サイズを取得する。
- `dpuSetInputTensorInCHWInt8()`: DPU タスクの入力テンソルを、INT8 形式で Caffe 表記順(チャネル/高さ/幅)で保存されたデータで設定する。
- `dpuSetInputTensorInCHWFP32()`: DPU タスクの入力テンソルを、FP32 形式で Caffe 表記順(チャネル/高さ/幅)で保存されたデータで設定する。
- `dpuSetInputTensorInHWCIint8()`: DPU タスクの入力テンソルを、INT8 形式で DPU 表記順(高さ/幅/チャネル)で保存されたデータで設定する。
- `dpuSetInputTensorInHWCfp32()`: DPU タスクの入力テンソルを、FP32 形式で DPU 表記順(チャネル/高さ/幅)で保存されたデータで設定する。
- `dpuGetOutputTensorInCHWInt8()`: DPU タスクの出力テンソルを取得し、INT8 形式で Caffe 表記順(チャネル/高さ/幅)で保存する。
- `dpuGetOutputTensorInCHWFP32()`: DPU タスクの出力テンソルを取得し、FP32 形式で Caffe 表記順(チャネル/高さ/幅)で保存する。
- `dpuGetOutputTensorInHWCIint8()`: DPU タスクの出力テンソルを取得し、INT8 形式で DPU 表記順(チャネル/高さ/幅)で保存する。
- `dpuGetOutputTensorInHWCfp32()`: DPU タスクの出力テンソルを取得し、FP32 形式で DPU 表記順(チャネル/高さ/幅)で保存する。
- `dpuRunSoftmax()`: 入力エレメントに対して softmax 計算を実行し、結果を出力メモリ バッファーに保存する。
- `dpuSetExceptionMode()`: エッジ DPU のランタイム N2Cube に対して例外処理モードを設定する。
- `dpuGetExceptionMode()`: ランタイム N2Cube の例外処理モードを取得する。
- `dpuGetExceptionMessage()`: N2Cube API で返されるエラー コード(常に負の値)からエラー メッセージを取得する。
- `dpuGetInputTotalSize()`: すべての境界入力テンソルを含む、DPU タスクの入力メモリ バッファーの合計サイズ(バイト)を取得する。
- `dpuGetOutputTotalSize()`: すべての境界出力テンソルを含む DPU タスクの出力メモリ バッファーの合計サイズ(バイト)を取得する。
- `dpuGetBoundaryIOTensor()`: 指定したテンソル名から DPU タスクの境界入力または出力テンソルを取得する。モデルがコンパイルされた後に VAI_C コンパイラによってテンソル名が一覧表示される。
- `dpuBindInputTensorBaseAddress()`: 入力メモリ バッファーの指定されたベース物理アドレスと仮想アドレスを DPU タスクに結び付ける。スプリット IO モードで VAI_C によってコンパイルされた DPU カーネルに対してのみ使用可能。スプリット IO モードで VAI_C によってコンパイルされた DPU カーネルに対してのみ使用できる。
- `dpuBindOutputTensorBaseAddress()`: 出力メモリ バッファーの指定された物理ベース アドレスと仮想ベース アドレスを DPU タスクに結び付けます。スプリット IO モードで VAI_C によってコンパイルされた DPU カーネルに対してのみ使用できる。

インクルード ファイル

n2cube.h

API のリスト

ライブラリ libn2cube 内の各 C++ API のプロトタイプとパラメーターについては、この後のセクションで詳しく説明します。

dpuOpen()

例

```
int dpuOpen()
```

引数

なし

説明

DPU リソースを使用する前に、「/dev/dpu」という DPU デバイス ファイルを接続して開く。

戻り値

成功した場合は 0、失敗した場合は負の値。エラーが生じた場合は、エラー メッセージ「Fail to open DPU device」が報告される。

関連資料

[dpuClose\(\)](#)

インクルード ファイル

n2cube.h

入力可能なインターフェイス

Vitis AI v1.0

[dpuClose\(\)](#)

例

```
int dpuClose()
```

引数

なし

説明

DPU リソースを使用した後、「/dev/dpu」という DPU デバイス ファイルとの接続を解除して閉じます。

戻り値

成功した場合は 0、失敗した場合は負のエラー ID 値。エラーが生じた場合は、エラー メッセージ「Fail to close DPU device」が報告される。

関連資料

[dpuOpen\(\)](#)

インクルード ファイル

n2cube.h

入力可能なインターフェイス

Vitis AI v1.0

dpuLoadKernel()

例

```
DPUKernel *dpuLoadKernel  
(  
const char *netName  
)
```

引数

- netName: ニューラル ネットワーク名へのポインター。ニューラル ネットワークのコンパイル後に、DNNC (Deep Neural Network Compiler) の VAI_C で生成された名前を使用する。DL アプリケーションごとに、CPU + DPU のハイブリッド バイナリ実行ファイルに多くの DPU カーネルが存在している可能性がある。各 DPU カーネルには、ほかと区別するために一意の名前が付いている。

説明

CPU + DPU のハイブリッド バイナリ実行ファイルから、指定されたニューラル ネットワークの DPU カーネルを DPU メモリ空間にロードする (カーネルの DPU 命令、重み、バイアスを含む)。

戻り値

成功した場合はロードされた DPU カーネルへのポインター。失敗した場合はエラーを報告する。

関連資料

[dpuDestroyKernel\(\)](#)

インクルード ファイル

n2cube.h

入力可能なインターフェイス

Vitis AI v1.0

dpuDestroyKernel()

例

```
Dint dpuDestroyKernel
(
DPUKernel *kernel
)
```

引数

- kernel: 無効にする DPU カーネルへのポインター。

説明

DPU カーネルを無効にして、関連するリソースを解放する。

戻り値

成功した場合は 0、失敗した場合はエラーを報告する。

関連資料

[dpuLoadKernel\(\)](#)

インクルード ファイル

n2cube.h

入力可能なインターフェイス

Vitis AI v1.0

dpuCreateTask()

例

```
int dpuCreateTask
(
DPUKernel *kernel,
int mode
);
```

引数

- kernel: 無効にする DPU カーネルへのポインター。
- mode: DPU タスクの動作モードで、3 つのモードがある。
 - T_MODE_NORMAL: デフォルト モードで、モード値「0」と同じ。

- T_MODE_PROF: DPU タスクの実行中にレイヤーごとにプロファイリング情報を生成する。これは性能分析に有効。
- T_MODE_DEBUG: デバッグとして、DPU タスクの CODE/BIAS/WEIGHT/INPUT/OUTPUT の生データをレイヤーごとにダンプする。

説明

DPU カーネルから DPU タスクをインスタンシエートし、対応する DPU メモリ バッファーを割り当てる。

戻り値

成功した場合は 0、失敗した場合はエラーを報告する。

インクルード ファイル

n2cube.h

入力可能なインターフェイス

Vitis AI v1.0

dpuDestroyTask()

書式

```
int dpuDestroyTask
(
    DPUTask *task
)
```

引数

- task:

破棄する DPU タスクへのポインター。

説明

DPU タスクを破棄し、関連リソースを解放します。

戻り値

正常に完了すると 0 が返され、それ以外の場合はエラーが報告されます。

関連 API

[dpuCreateTask\(\)](#)

インクルード ファイル

n2cube.h

対応バージョン

Vitis AI v1.0

dpuRunTask()

例

```
int dpuRunTask
(
    DPUTask *task
);
```

引数

- task: DPU タスクへのポインター。

説明

DPU タスクの実行を開始する。

戻り値

成功した場合は 0、失敗した場合は負の値。

インクルード ファイル

n2cube.h

入力可能なインターフェイス

Vitis AI v1.0

dpuSetTaskPriority()

例

```
int dpuSetTaskPriority
(
    DPUTask *task,
    uint8_t priority
);
```

引数

- task: DPU タスクへのポインター。
- priority: DPU タスクに指定される優先順位。範囲は 0 (最も高い優先度) ~ 15 (最も低い優先度)。

説明

実行時に DPU タスクの優先度を指定した値に動的に設定する。優先度の範囲は 0 (最も高い優先度) ~ 15 (最も低い優先度)。指定しない場合、DPU タスクの優先度はデフォルトの 15 に設定される。

戻り値

成功した場合は 0、失敗した場合は負の値。

関連資料

[dpuGetTaskPriority\(\)](#)

インクルード ファイル

n2cube.h

入力可能なインターフェイス

Vitis AI v1.0

dpuGetTaskPriority()

例

```
uint8_t
    dpuGetTaskPriority
(
    DPUTask *task
);
```

引数

- task: DPU タスクへのポインター。

説明

DPU タスクの優先度を取得する。デフォルトは 15。

戻り値

成功した場合は DPU タスクの優先度の値。失敗した場合は 0xFF。

関連資料

[dpuSetTaskPriority\(\)](#)

インクルード ファイル

n2cube.h

入力可能なインターフェイス

Vitis AI v1.0

dpuSetTaskAffinity()

例

```
int
    dpuSetTaskAffinity
(
    DPUTask *task,
    uint32_t coreMask
);
```

引数

- task: DPU タスクへのポインター。
- coreMask: 指定する DPU コア マスク。各ビットが 1 つの DPU コアを表します(最下位ビットはコア 0、2 番目に最下位ビットはコア 1...と続く)。一度に複数のマスク ビットを指定できますが、有効な最大コア数を超えることはできません。たとえば、マスク値 0x3 は、タスクを DPU コア 0 および 1 に割り当てる能够性を示し、コア 0 または 1 のいずれかが有効であれば、すぐにスケジューリングされます。

説明

実行時に DPU コアにおける DPU タスクのアフィニティを動的に設定します。これにより、ユーザーは特定の要件を満たすために DPU コアの割り当てとスケジューリングに柔軟に対応できるようになります。指定しない場合、実行時に DPU タスクを有効な任意の DPU コアに割り当てる能够性があります。

戻り値

成功した場合は 0、失敗した場合は負の値。

関連資料

[dpuGetTaskAffinity \(\)](#)

インクルード ファイル

n2cube.h

入力可能なインターフェイス

Vitis AI v1.0

dpuGetTaskAffinity ()

例

```
uint32_t dpuGetTaskAffinity
(
    DPUTask *task
);
```

引数

- task: DPU タスクへのポインター。

説明

DPU コアにおける DPU タスクのアフィニティを取得します。指定しない場合、DPU タスクはデフォルトで有効なすべての DPU コアに割り当て可能です。たとえば、ターゲットシステムに 3 つの DPU コアがある場合、アフィニティは 0x7 となります。

戻り値

成功した場合は DPU コアのアフィニティ マスク ビット。失敗した場合は 0。

関連資料

[dpuSetTaskAffinity\(\)](#)

インクルード ファイル

n2cube.h

入力可能なインターフェイス

Vitis AI v1.0

dpuEnableTaskProfile()

例

```
int dpuEnableTaskProfile
(
    DPUTask *task
);
```

引数

- task: DPU タスクへのポインター。

説明

プロファイリング モードで DPU タスクを設定します。プロファイリング機能は、デバッグ モードで VAI_C によって生成された DPU カーネルに対してのみ有効です。

戻り値

成功した場合は 0、失敗した場合はエラーを報告する。

関連資料

[dpuCreateTask\(\)](#)

[dpuEnableTaskDebug\(\)](#)

インクルード ファイル

n2cube.h

入力可能なインターフェイス

Vitis AI v1.0

dpuEnableTaskDebug()

例

```
int dpuEnableTaskDebug
(
    DPUTask *task
);
```

引数

- task: DPU タスクへのポインター。

説明

ダンプ モードで DPU タスクを設定します。ダンプ機能は、デバッグ モードで VAI_C によって生成された DPU カーネルに対してのみ有効です。

戻り値

成功した場合は 0、失敗した場合はエラーを報告する。

関連資料

[dpuCreateTask\(\)](#)

[dpuEnableTaskProfile\(\)](#)

インクルード ファイル

n2cube.h

入力可能なインターフェイス

Vitis AI v1.0

dpuGetTaskProfile()

例

```
int dpuGetTaskProfile
(
    DPUTask *task
);
```

引数

- task: DPU タスクへのポインター。

説明

実行後の DPU タスクの実行時間 (us) を取得する。

戻り値

実行後の DPU タスクの実行時間 (us)。

関連資料

[dpuGetNodeProfile\(\)](#)

インクルード ファイル

n2cube.h

入力可能なインターフェイス

Vitis AI v1.0

dpuGetNodeProfile()

例

```
int dpuGetNodeProfile
(
    DPUTask *task,
    const char*nodeName
);
```

引数

- task: DPU タスクへのポインター。
- nodeName: DPU ノード名へのポインター。

説明

DPU タスクの実行が完了した後、DPU ノードの実行時間 (us) を取得します。

戻り値

DPU タスク実行後の DPU ノードの実行時間 (us)。この機能は、デバッグ モードで VAI_C によって生成された DPU カーネルに対してのみ有効です。

関連資料

[dpuGetTaskProfile\(\)](#)

インクルード ファイル

n2cube.h

入力可能なインターフェイス

Vitis AI v1.0

dpuGetInputTensorCnt()

書式

```
Int dpuGetInputTensorCnt
(
    DPUTask *task,
    const char*nodeName
);
```

引数

- task: DPU タスクへのポインター。
- nodeName: DPU ノード名へのポインター。

注記: DPU カーネルやタスクの入力ノード名は、ニューラル ネットワークが VAI_C によってコンパイルされた後にリストされます。無効なノード名を指定すると、エラー メッセージが報告されます。

説明

1 つの DPU タスクの指定されたノードに対する入力テンソルの総数を取得します。

戻り値

指定したノードの入力テンソルの総数。

関連 API

[dpuGetOutputTensorCnt\(\)](#)

インクルード ファイル

n2cube.h

対応バージョン

Vitis AI v1.0

dpuGetInputTensor()

書式

```
DPUTensor*dpuGetInputTensor
(
    DPUTask *task,
    const char*nodeName,
    int idx = 0
);
```

引数

- task: DPU タスクへのポインター。
- nodeName: DPU ノード名へのポインター。

注記: DPU カーネルやタスクの入力ノード名は、ニューラル ネットワークが VAI_C によってコンパイルされた後にリストされます。無効なノード名を指定すると、エラー メッセージが報告されます。

- idx:

ノードの入力テンソルのいずれか 1 つをインデックスで指定します (デフォルト値は 0)。

説明

DPU タスクの入力テンソルを取得します。

戻り値

正常に完了すると、タスクの入力テンソルへのポインターが返され、それ以外の場合はエラーが報告されます。

関連 API

[dpuGetOutputTensor\(\)](#)

インクルード ファイル

n2cube.h

対応バージョン

Vitis AI v1.0

dpuGetInputTensorAddress()

書式

```
int8_t*
    dpuGetInputTensorAddress
(
DPUTask *task,
const char*nodeName,
int idx = 0
);
```

引数

- task: DPU タスクへのポインター。
- nodeName: DPU ノード名へのポインター。

注記: DPU カーネルやタスクの入力ノード名は、ニューラル ネットワークが VAI_C によってコンパイルされた後にリストされます。無効なノード名を指定すると、エラー メッセージが報告されます。

- idx:

ノードの入力テンソルのいずれか 1 つをインデックスで指定します (デフォルト値は 0)。

説明

DPU タスクの入力テンソルの開始アドレスを取得します。

戻り値

正常に完了すると、タスクの入力テンソルの開始アドレスが返され、それ以外の場合はエラーが報告されます。

関連 API

[dpuGetOutputTensorAddress\(\)](#)

インクルード ファイル

n2cube.h

対応バージョン

Vitis AI v1.0

dpuGetInputTensorSize()

書式

```
int dpuGetInputTensorSize
(
    DPUTask *task,
    const char*nodeName,
    int idx = 0
);
```

引数

- task: DPU タスクへのポインター。
- nodeName: DPU ノード名へのポインター。

注記: DPU カーネルやタスクの入力ノード名は、ニューラル ネットワークが VAI_C によってコンパイルされた後にリストされます。無効なノード名を指定すると、エラー メッセージが報告されます。

- idx:

ノードの入力テンソルのいずれか 1 つをインデックスで指定します (デフォルト値は 0)。

説明

DPU タスクの入力テンソルのサイズ (単位: バイト) を取得します。

戻り値

正常に完了すると、タスクの入力テンソルのサイズが返され、それ以外の場合はエラーが報告されます。

関連 API

[dpuGetOutputTensorSize\(\)](#)

インクルード ファイル

n2cube.h

対応バージョン

Vitis AI v1.0

dpuGetInputTensorScale()

書式

```
float dpuGetInputTensorScale
(
    DPUTask *task,
    const char*nodeName,
    int idx = 0
);
```

引数

- task: DPU タスクへのポインター。
- nodeName: DPU ノード名へのポインター。

注記: DPU カーネルやタスクの出力ノード名は、ニューラル ネットワークが VAI_C によってコンパイルされた後にリストされます。無効なノード名を指定すると、エラー メッセージが報告されます。

- idx: ノードの单一出力テンソルのインデックス。デフォルト値は 0。

説明

DPU タスクの入力テンソルのスケール値を取得します。各 DPU 入力テンソルには、データ型を INT8 と FP32 で変換する際の量子化情報を示すスケール値が 1 つあります。

戻り値

正常に完了すると、タスクの入力テンソルのスケール値が返され、それ以外の場合はエラーが報告されます。

関連 API

[dpuGetOutputTensorScale\(\)](#)

インクルード ファイル

n2cube.h

対応バージョン

Vitis AI v1.0

dpuGetInputTensorHeight()

書式

```
int dpuGetInputTensorHeight
(
    DPUTask *task,
    const char*nodeName,
    int idx = 0
);
```

引数

- task: DPU タスクへのポインター。
- nodeName: DPU ノード名へのポインター。

注記: DPU カーネルやタスクの出力ノード名は、ニューラル ネットワークが VAI_C によってコンパイルされた後にリストされます。無効なノード名を指定すると、エラー メッセージが報告されます。

- idx: ノードの单一出力テンソルのインデックス。デフォルト値は 0。

説明

DPU タスクの入力テンソルの次元の高さ (height) を取得します。

戻り値

正常に完了すると、タスクの入力テンソルの次元の高さ (height) が返され、それ以外の場合はエラーが報告されます。

関連 API

[dpuGetInputTensorWidth\(\)](#)
[dpuGetInputTensorChannel\(\)](#)
[dpuGetOutputTensorHeight\(\)](#)
[dpuGetOutputTensorWidth\(\)](#)
[dpuGetOutputTensorChannel\(\)](#)

インクルード ファイル

n2cube.h

対応バージョン

Vitis AI v1.0

dpuGetInputTensorWidth()

書式

```
int dpuGetInputTensorWidth
(
    DPUTask *task,
    const char*nodeName,
    int idx = 0
);
```

引数

- task: DPU タスクへのポインター。
- nodeName: DPU ノード名へのポインター。

注記: DPU カーネルやタスクの出力ノード名は、ニューラル ネットワークが VAI_C によってコンパイルされた後にリストされます。無効なノード名を指定すると、エラー メッセージが報告されます。

- idx: ノードの单一出力テンソルのインデックス。デフォルト値は 0。

説明

DPU タスクの入力テンソルの次元の幅 (width) を取得します。

戻り値

正常に完了すると、タスクの入力テンソルの次元の幅 (width) が返され、それ以外の場合はエラーが報告されます。

関連 API

[dpuGetInputTensorHeight\(\)](#)
[dpuGetInputTensorChannel\(\)](#)
[dpuGetOutputTensorHeight\(\)](#)
[dpuGetOutputTensorWidth\(\)](#)
[dpuGetOutputTensorChannel\(\)](#)

インクルード ファイル

n2cube.h

対応バージョン

Vitis AI v1.0

dpuGetInputTensorChannel()

書式

```
int dpuGetInputTensorChannel
(
    DPUTask *task,
    const char*nodeName,
    int idx = 0
);
```

引数

- task: DPU タスクへのポインター。
- nodeName: DPU ノード名へのポインター。

注記: DPU カーネルやタスクの出力ノード名は、ニューラル ネットワークが VAI_C によってコンパイルされた後にリストされます。無効なノード名を指定すると、エラー メッセージが報告されます。

- idx: ノードの单一出力テンソルのインデックス。デフォルト値は 0。

説明

DPU タスクの入力テンソルの次元のチャネル (channel) を取得します。

戻り値

正常に完了すると、タスクの入力テンソルの次元のチャネル (channel) が返され、それ以外の場合はエラーが報告されます。

関連 API

[dpuGetInputTensorHeight\(\)](#)

[dpuGetInputTensorWidth\(\)](#)

[dpuGetOutputTensorHeight\(\)](#)

[dpuGetOutputTensorWidth\(\)](#)

[dpuGetOutputTensorChannel\(\)](#)

インクルード ファイル

n2cube.h

対応バージョン

Vitis AI v1.0

dpuGetOutputTensorCnt()

書式

```
Int dpuGetOutputTensorCnt
(
    DPUTask *task,
    const char*nodeName
);
```

引数

- task: DPU タスクへのポインター。
- nodeName: DPU ノード名へのポインター。

注記: DPU カーネルやタスクの出力ノード名は、ニューラル ネットワークが VAI_C によってコンパイルされた後にリストされます。無効なノード名を指定すると、エラー メッセージが報告されます。

- idx: ノードの単一出力テンソルのインデックス。デフォルト値は 0。

説明

1 つの DPU タスクの指定されたノードに対する出力テンソルの総数を取得する。

戻り値

DPU タスクの出力テンソルの総数。

関連 API

[dpuGetInputTensorCnt\(\)](#)

インクルード ファイル

n2cube.h

対応バージョン

Vitis AI v1.0

dpuGetOutputTensor()

書式

```
DPUTensor*dpuGetOutputTensor
(
    DPUTask *task,
    const char*nodeName,
    int idx = 0
);
```

引数

- task: DPU タスクへのポインター。

- nodeName: DPU ノード名へのポインター。

注記: DPU カーネルやタスクの出力ノード名は、ニューラル ネットワークが VAI_C によってコンパイルされた後にリストされます。無効なノード名を指定すると、エラー メッセージが報告されます。

- idx: ノードの单一出力テンソルのインデックス。デフォルト値は 0。

説明

DPU タスクの出力テンソルを取得します。

戻り値

正常に完了すると、タスクの出力テンソルへのポインターが返され、それ以外の場合はエラーが報告されます。

関連 API

[dpuGetInputTensor\(\)](#)

インクルード ファイル

n2cube.h

対応バージョン

Vitis AI v1.0

dpuGetOutputTensorAddress()

書式

```
int8_t* dpuGetOutputTensorAddress
(
    DPUTask *task,
    const char*nodeName,
    int idx = 0
);
```

引数

- task: DPU タスクへのポインター。
- nodeName: DPU ノード名へのポインター。

注記: DPU カーネルやタスクの出力ノード名は、ニューラル ネットワークが VAI_C によってコンパイルされた後にリストされます。無効なノード名を指定すると、エラー メッセージが報告されます。

- idx: ノードの单一出力テンソルのインデックス。デフォルト値は 0。

説明

DPU タスクの出力テンソルの開始アドレスを取得します。

戻り値

正常に完了すると、タスクの出力テンソルの開始アドレスが返され、それ以外の場合はエラーが報告されます。

関連 API

[dpuGetInputTensorAddress\(\)](#)

インクルード ファイル

n2cube.h

対応バージョン

Vitis AI v1.0

dpuGetOutputTensorSize()

書式

```
int dpuGetOutputTensorSize
(
    DPUTask *task,
    const char*nodeName,
    int idx = 0
);
```

引数

- task: DPU タスクへのポインター。
- nodeName: DPU ノード名へのポインター。

注記: DPU カーネルやタスクの出力ノード名は、ニューラル ネットワークが VAI_C によってコンパイルされた後にリストされます。無効なノード名を指定すると、エラー メッセージが報告されます。

- idx: ノードの单一出力テンソルのインデックス。デフォルト値は 0。

説明

DPU タスクの出力テンソルのサイズ (単位: バイト) を取得します。

戻り値

正常に完了すると、タスクの出力テンソルのサイズが返され、それ以外の場合はエラーが報告されます。

関連 API

[dpuGetInputTensorSize\(\)](#)

インクルード ファイル

n2cube.h

対応バージョン

Vitis AI v1.0

dpuGetOutputTensorScale()

書式

```
float dpuGetOutputTensorScale
(
    DPUTask *task,
    const char *nodeName,
    int idx = 0
);
```

引数

- task: DPU タスクへのポインター。
- nodeName: DPU ノード名へのポインター。

注記: DPU カーネルやタスクの出力ノード名は、ニューラル ネットワークが VAI_C によってコンパイルされた後にリストされます。無効なノード名を指定すると、エラー メッセージが報告されます。

- idx: ノードの単一出力テンソルのインデックス。デフォルト値は 0。

説明

DPU タスクの出力テンソルのスケール値を取得します。各 DPU 出力テンソルには、データ型を INT8 と FP32 で変換する際の量子化情報を示すスケール値が 1 つあります。

戻り値

正常に完了すると、タスクの出力テンソルのスケール値が返され、それ以外の場合はエラーが報告されます。

関連 API

[dpuGetInputTensorScale\(\)](#)

インクルード ファイル

n2cube.h

対応バージョン

Vitis AI v1.0

dpuGetOutputTensorHeight()

書式

```
int dpuGetOutputTensorHeight
(
    DPUTask *task,
    const char*nodeName,
    int idx = 0
);
```

引数

- task: DPU タスクへのポインター。
- nodeName: DPU ノード名へのポインター。

注記: DPU カーネルやタスクの出力ノード名は、ニューラル ネットワークが VAI_C によってコンパイルされた後にリストされます。無効なノード名を指定すると、エラー メッセージが報告されます。

- idx: ノードの单一出力テンソルのインデックス。デフォルト値は 0。

説明

DPU タスクの出力テンソルの次元の高さ (height) を取得します。

戻り値

正常に完了すると、タスクの出力テンソルの次元の高さ (height) が返され、それ以外の場合はエラーが報告されます。

関連 API

[dpuGetOutputTensorWidth\(\)](#)

[dpuGetOutputTensorChannel\(\)](#)

[dpuGetInputTensorHeight\(\)](#)

[dpuGetInputTensorWidth\(\)](#)

[dpuGetInputTensorChannel\(\)](#)

インクルード ファイル

n2cube.h

対応バージョン

Vitis AI v1.0

dpuGetOutputTensorWidth()

例

```
int dpuGetOutputTensorWidth
(
DPUTask *task,
const char*nodeName,
int idx = 0
);
```

引数

- task: DPU タスクへのポインター。
- nodeName: DPU ノード名へのポインター。

注記: DPU カーネルやタスクの出力ノード名は、ニューラル ネットワークが VAI_C によってコンパイルされた後にリストされます。無効なノード名が指定された場合、エラー メッセージが報告されます。

- idx: ノードの单一出力テンソルのインデックス。デフォルト値は 0。

説明

DPU タスクの出力テンソルの幅を取得する。

戻り値

成功した場合はタスクの出力テンソルの幅の値。失敗した場合はエラーが報告される。

関連資料

[dpuGetOutputTensorHeight\(\)](#)

[dpuGetOutputTensorChannel\(\)](#)

[dpuGetInputTensorHeight\(\)](#)

[dpuGetInputTensorWidth\(\)](#)

[dpuGetInputTensorChannel\(\)](#)

インクルード ファイル

n2cube.h

入力可能なインターフェイス

Vitis AI v1.0

dpuGetOutputTensorChannel()

例

```
int dpuGetOutputTensorChannel
(
    DPUTask *task,
    const char*nodeName,
    int idx = 0
);
```

引数

- task: DPU タスクへのポインター。
- nodeName: DPU ノード名へのポインター。

注記: DPU カーネルやタスクの出力ノード名は、ニューラル ネットワークが VAI_C によってコンパイルされた後にリストされます。無効なノード名が指定された場合、エラー メッセージが報告されます。

- idx: ノードの单一出力テンソルのインデックス。デフォルト値は 0。

説明

DPU タスクの出力テンソルのチャネル サイズを取得する。

戻り値

成功した場合はタスクの出力テンソルのチャネル サイズの値。失敗した場合はエラーが報告される。

関連資料

[dpuGetOutputTensorHeight\(\)](#)

[dpuGetOutputTensorWidth\(\)](#)

[dpuGetInputTensorHeight\(\)](#)

[dpuGetInputTensorWidth\(\)](#)

[dpuGetInputTensorChannel\(\)](#)

インクルード ファイル

n2cube.h

入力可能なインターフェイス

Vitis AI v1.0

dpuGetTensorAddress()

例

```
int dpuGetTensorAddress
(
    DPUTensor* tensor
);
```

引数

- tensor: DPU テンソルへのポインター。

説明

DPU テンソルの開始アドレスを取得します。

戻り値

テンソルの開始アドレス。失敗した場合はエラーが報告される。

関連資料

[dpuGetInputTensorAddress\(\)](#)

[dpuGetOutputTensorAddress\(\)](#)

インクルード ファイル

n2cube.h

入力可能なインターフェイス

Vitis AI v1.0

dpuGetTensorSize()

例

```
int dpuGetTensorSize
(
    DPUTensor* tensor
);
```

引数

- tensor: DPU テンソルへのポインター。

説明

DPU テンソルのサイズ (バイト) を取得する。

戻り値

テンソルのサイズ。失敗した場合はエラーが報告される。

関連資料

[dpuGetInputTensorSize\(\)](#)

[dpuGetOutputTensorSize\(\)](#)

インクルード ファイル

n2cube.h

入力可能なインターフェイス

Vitis AI v1.0

dpuGetTensorScale()

例

```
float dpuGetTensorScale
(
    DPUTensor* tensor
);
```

引数

- tensor: DPU テンソルへのポインター。

説明

DPU テンソルのスケール値を取得します。

戻り値

テンソルのスケール値。失敗した場合はエラーが報告される。ユーザーは、このスケール係数を使用して、DPU 入力テンソルの量子化 (Float32 から Int8) または DPU 出力テンソルの逆量子化 (Int8 から Float32) を実行できます。

関連資料

[dpuGetInputTensorScale\(\)](#)

[dpuGetOutputTensorScale\(\)](#)

インクルード ファイル

n2cube.h

入力可能なインターフェイス

Vitis AI v1.0

dpuGetTensorHeight()

例

```
float dpuGetTensorHeight
(
    DPUTensor* tensor
);
```

引数

- tensor: DPU テンソルへのポインター。

説明

DPU テンソルの高さを取得する。

戻り値

テンソルの高さ。失敗した場合はエラーが報告される。

関連資料

[dpuGetInputTensorHeight\(\)](#)

[dpuGetOutputTensorHeight\(\)](#)

インクルード ファイル

n2cube.h

入力可能なインターフェイス

Vitis AI v1.0

dpuGetTensorWidth()

例

```
float dpuGetTensorWidth
(
    DPUTensor* tensor
);
```

引数

- tensor: DPU テンソルへのポインター。

説明

DPU テンソルの幅を取得します。

戻り値

テンソルの幅。失敗した場合はエラーが報告される。

関連資料

[dpuGetInputTensorWidth\(\)](#)

[dpuGetOutputTensorWidth\(\)](#)

インクルード ファイル

n2cube.h

入力可能なインターフェイス

Vitis AI v1.0

dpuGetTensorChannel()

例

```
float dpuGetTensorChannel
(
    DPUTensor* tensor
);
```

引数

- tensor: DPU テンソルへのポインター。

説明

DPU テンソルのチャネル サイズを取得する。

戻り値

テンソルのチャネル サイズの値。失敗した場合はエラーが報告される。

関連資料

[dpuGetInputTensorChannel\(\)](#)

[dpuGetOutputTensorChannel\(\)](#)

インクルード ファイル

n2cube.h

入力可能なインターフェイス

Vitis AI v1.0

dpuSetInputTensorInCHWInt8()

例

```
int dpuSetInputTensorInCHWInt8
(
    DPUTask *task,
    const char *nodeName,
    int8_t *data,
    int size,
    int idx = 0
)
```

引数

- task: DPU タスクへのポインター。
- nodeName: DPU ノード名へのポインター。
- data: 入力データの開始アドレスへのポインター。
- size: 設定される入力データのサイズ (バイト単位)。
- idx: ノードの单一入力テンソルのインデックス。デフォルト値は 0。

説明

DPU タスクの入力テンソルを CPU メモリ ブロックからのデータを使用して設定します。データ型 INT8 で、Caffe ブロブの順序 (チャネル、高さ、重み) で保存されます。

戻り値

成功した場合は 0、失敗した場合はエラーが報告される。

関連資料

[dpuSetInputTensorInCHWF32\(\)](#)

[dpuSetInputTensorInHWCInt8\(\)](#)

[dpuSetInputTensorInHWCFP32\(\)](#)

インクルード ファイル

n2cube.h

入力可能なインターフェイス

Vitis AI v1.0

dpuSetInputTensorInCHWFP32()

例

```
int dpuSetInputTensorInCHWFP32
(
    DPUTask *task,
    const char *nodeName,
    float *data,
    int size,
    int idx = 0
)
```

引数

- task: DPU タスクへのポインター。
- nodeName: DPU ノード名へのポインター。
- data: 入力データの開始アドレスへのポインター。
- size: 設定される入力データのサイズ (バイト単位)。
- idx: ノードの单一入力テンソルのインデックス。デフォルト値は 0。

説明

DPU タスクの入力テンソルを CPU メモリ ブロックからのデータを使用して設定します。データ型は 32 ビットの float で DPU テンソルの順序 (高さ、重み、チャネル) で保存されます。

戻り値

成功した場合は 0、失敗した場合はエラーが報告される。

関連資料

[dpuSetInputTensorInCHWInt8\(\)](#)
[dpuSetInputTensorInHWCInt8\(\)](#)
[dpuSetInputTensorInHWCFP32\(\)](#)

インクルード ファイル

n2cube.h

入力可能なインターフェイス

Vitis AI v1.0

dpuSetInputTensorInHWCInt8()

例

```
int dpuSetInputTensorInHWCInt8
(
    DPUTask *task,
    const char *nodeName,
    int8_t *data,
    int size,
    int idx = 0
)
```

引数

- task: DPU タスクへのポインター。
- nodeName: DPU ノード名へのポインター。
- data: 入力データの開始アドレスへのポインター。
- size: 設定される入力データのサイズ (バイト単位)。
- idx: ノードの单一入力テンソルのインデックス。デフォルト値は 0。

説明

DPU タスクの入力テンソルを CPU メモリ ブロックからのデータを使用して設定します。データ型は 32 ビットの float で DPU テンソルの順序 (高さ、重み、チャネル) で保存されます。

戻り値

成功した場合は 0、失敗した場合はエラーが報告される。

関連資料

[dpuSetInputTensorInCHWInt8\(\)](#)

[dpuSetInputTensorInCHWF32\(\)](#)

[dpuSetInputTensorInHWCFP32\(\)](#)

インクルード ファイル

n2cube.h

入力可能なインターフェイス

Vitis AI v1.0

dpuSetInputTensorInHWCFP32()

例

```
int dpuSetInputTensorInHWCFP32
(
    DPUTask *task,
    const char *nodeName,
    float *data,
    int size,
    int idx = 0
)
```

引数

- task: DPU タスクへのポインター。
- nodeName: DPU ノード名へのポインター。
- data: 入力データの開始アドレスへのポインター。
- size: 設定される入力データのサイズ (バイト単位)。
- idx: ノードの单一入力テンソルのインデックス。デフォルト値は 0。

説明

DPU タスクの入力テンソルを CPU メモリ ブロックからのデータを使用して設定します。データ型は 32 ビットの float で DPU テンソルの順序 (高さ、重み、チャネル) で保存されます。

戻り値

成功した場合は 0、失敗した場合はエラーが報告される。

関連資料

[dpuSetInputTensorInCHWInt8\(\)](#)

[dpuSetInputTensorInCHWFP32\(\)](#)

[dpuSetInputTensorInHWCInt8\(\)](#)

インクルード ファイル

n2cube.h

入力可能なインターフェイス

Vitis AI v1.0

dpuGetOutputTensorInCHWInt8()

例

```
int dpuGetOutputTensorInCHWInt8
(
    DPUTask *task,
    const char *nodeName,
    int8_t *data,
    int size,
    int idx = 0
)
```

引数

- task: DPU タスクへのポインター。
- nodeName: DPU ノード名へのポインター。
- data: 出力テンソルのデータを格納するための CPU メモリ ブロックの開始アドレス。
- size: 格納される出力データのサイズ (バイト単位)。
- idx: ノードの单一出力テンソルのインデックス。デフォルト値は 0。

説明

DPU タスクの出力テンソルを取得して、CPU メモリ ブロックに格納します。データ型は INT8 で DPU テンソルの順序 (高さ、重み、チャネル) で格納されます。

戻り値

成功した場合は 0、失敗した場合はエラーが報告される。

関連資料

[dpuGetOutputTensorInCHWF32\(\)](#)

[dpuGetOutputTensorInHWCInt8\(\)](#)

[dpuGetOutputTensorInHWCFP32\(\)](#)

インクルード ファイル

n2cube.h

入力可能なインターフェイス

Vitis AI v1.0

dpuGetOutputTensorInCHWFP32()

例

```
int dpuGetOutputTensorInCHWFP32
(
    DPUTask *task,
    const char *nodeName,
    float *data,
    int size,
    int idx = 0
)
```

引数

- task: DPU タスクへのポインター。
- nodeName: DPU ノード名へのポインター。
- data: 出力テンソルのデータを格納するための CPU メモリ ブロックの開始アドレス。
- size: 格納される出力データのサイズ (バイト単位)。
- idx: ノードの单一出力テンソルのインデックス。デフォルト値は 0。

説明

DPU タスクの出力テンソルを取得して、CPU メモリ ブロックに格納します。これらのデータは、32 ビットの float 型で、Caffe プロトの順序 (チャネル、高さ、重み) で格納されます。

戻り値

成功した場合は 0、失敗した場合はエラーが報告される。

関連資料

[dpuGetOutputTensorInCHWInt8\(\)](#)

[dpuGetOutputTensorInHWCInt8\(\)](#)

[dpuGetOutputTensorInHWCFP32\(\)](#)

インクルード ファイル

n2cube.h

入力可能なインターフェイス

Vitis AI v1.0

dpuGetOutputTensorInHWCIInt8()

例

```
int dpuGetOutputTensorInHWCIInt8
(
    DPUTask *task,
    const char *nodeName,
    int8_t *data,
    int size,
    int idx = 0
)
```

引数

- task: DPU タスクへのポインター。
- nodeName: DPU ノード名へのポインター。
- data: 出力テンソルのデータを格納するための CPU メモリ ブロックの開始アドレス。
- size: 格納される出力データのサイズ (バイト単位)。
- idx: ノードの单一出力テンソルのインデックス。デフォルト値は 0。

説明

DPU タスクの出力テンソルを取得して、CPU メモリ ブロックに格納します。データ型は INT8 で DPU テンソルの順序 (高さ、重み、チャネル) で格納されます。

戻り値

成功した場合は 0、失敗した場合はエラーが報告される。

関連資料

[dpuGetOutputTensorInCHWInt8\(\)](#)

[dpuGetOutputTensorInCHWFP32\(\)](#)

[dpuGetOutputTensorInHWCFP32\(\)](#)

インクルード ファイル

n2cube.h

入力可能なインターフェイス

Vitis AI v1.0

dpuGetOutputTensorInHWCFP32()

例

```
int dpuGetOutputTensorInHWCFP32
(
    DPUTask *task,
    const char *nodeName,
    float *data,
    int size,
    int idx = 0
)
```

引数

- task: DPU タスクへのポインター。
- nodeName: DPU ノード名へのポインター。
- data: 出力テンソルのデータを格納するための CPU メモリ ブロックの開始アドレス。
- size: 格納される出力データのサイズ (バイト単位)。
- idx: ノードの单一出力テンソルのインデックス。デフォルト値は 0。

説明

DPU タスクの出力テンソルを取得して、CPU メモリ ブロックに格納します。データ型は 32 ビットの float で DPU テンソルの順序 (高さ、重み、チャネル) で格納されます。

戻り値

成功した場合は 0、失敗した場合はエラーが報告される。

関連資料

[dpuGetOutputTensorInCHWInt8\(\)](#)
[dpuGetOutputTensorInCHWFP32\(\)](#)
[dpuGetOutputTensorInHWCInt8\(\)](#)

インクルード ファイル

n2cube.h

入力可能なインターフェイス

Vitis AI v1.0

dpuRunSoftmax()

例

```
int dpuRunSoftmax
(
    int8_t *input,
    float *output,
    int numClasses,
    int batchSize,
    float scale
)
```

引数

- input: int8_t 型の softmax 入力エレメントを格納するためのポインター。
- output: softmax の実行結果 (浮動小数点型) を格納するためのポインター。このメモリ空間は、caller 関数によって割り当ておよび管理される必要があります。
- numClasses: softmax 計算が適用されるクラス数。
- batchSize: softmax 計算のバッチ サイズ。このパラメーターは、numClasses の入力値でエレメント数を分割した値を指定する必要があります。
- scale: softmax 計算の前に入力エレメントに適用されるスケール値。このパラメーターは、通常、API dpuGetTensorScale() を使用して取得できる。

説明

入力エレメントに対して softmax 計算を実行し、結果を出力メモリ バッファーに保存します。この API は、ハードウェア実装のモジュールを利用できる場合には、高速化のために DPU コアを活用します。「dexplorer -w」を実行して、DPU シグネチャ情報を表示します。

戻り値

成功した場合は 0。

インクルード ファイル

n2cube.h

入力可能なインターフェイス

Vitis AI v1.0

dpuSetExceptionMode()

例

```
int dpuSetExceptionMode
(
    int mode
)
```

引数

- mode: 指定するランタイム N2Cube の例外処理モード。有効な値は次のとおり。
 - N2CUBE_EXCEPTION_MODE_PRINT_AND_EXIT
 - N2CUBE_EXCEPTION_MODE_RET_ERR_CODE

説明

エッジ DPU のランタイム N2Cube に対して例外処理モードを設定します。libn2cube ライブラリに含まれるすべての API に影響します。

N2CUBE_EXCEPTION_MODE_PRINT_AND_EXIT が指定されている場合、エラーが発生すると、呼び出された N2Cube API はエラー メッセージを出力し、DPU アプリケーションの実行を終了します。これは N2Cube API のデフォルト モードです。

N2CUBE_EXCEPTION_MODE_RET_ERR_CODE が指定されている場合、呼び出された N2Cube API はエラーが生じた場合にのみエラー コードを返します。呼び出し元が API dpuGetExceptionMessage() を使用してエラー メッセージを記録したり、リソースを解放するなど、次に示すような例外処理プロセスを管理する必要があります。

戻り値

成功した場合は 0、失敗した場合は負の値。

関連資料

[dpuGetExceptionMode\(\)](#)

[dpuGetExceptionMessage](#)

インクルード ファイル

n2cube.h

入力可能なインターフェイス

Vitis AI v1.0

dpuGetExceptionMode()

例

```
int dpuGetExceptionMode()
```

引数

なし

説明

ランタイム N2Cube の例外処理モードを取得する。

戻り値

N2Cube API の現在の例外処理モード。

有効な値:

- N2CUBE_EXCEPTION_MODE_PRINT_AND_EXIT
- N2CUBE_EXCEPTION_MODE_RET_ERR_CODE

関連資料

[dpuSetExceptionMode\(\)](#)

[dpuGetExceptionMessage](#)

インクルード ファイル

n2cube.h

入力可能なインターフェイス

Vitis AI v1.0

dpuGetExceptionMessage

例

```
const char *dpuGetExceptionMessage
(
int error_code
)
```

引数

- error code: N2Cube API が返すエラー コード。

説明

N2Cube API で返されるエラー コード (常に負の値) からエラー メッセージを取得する。

戻り値

const 文字列へのポインター。error_code のエラー メッセージを示す。

関連資料

[dpuSetExceptionMode\(\)](#)

[dpuGetExceptionMode\(\)](#)

インクルード ファイル

n2cube.h

入力可能なインターフェイス

Vitis AI v1.0

dpuGetInputTotalSize()

例

```
int dpuGetInputTotalSize
(
    DPUTask *task,
)
```

引数

- task: DPU タスクへのポインター。

説明

すべての境界入力テンソルを含む、DPU タスクの入力メモリ バッファーの合計サイズ (バイト) を取得します。

戻り値

DPU タスクのすべての境界入力テンソルを含む合計サイズ (バイト)。

関連資料

[dpuGetOutputTotalSize\(\)](#)

インクルード ファイル

n2cube.h

入力可能なインターフェイス

Vitis AI v1.0

dpuGetOutputTotalSize()

例

```
int dpuGetOutputTotalSize
(
    DPUTask *task,
)
```

引数

- task: DPU タスクへのポインター。

説明

すべての境界出力テンソルを含む、DPU タスクの出力メモリ バッファーの合計サイズ (バイト) を取得します。

戻り値

DPU タスクのすべての境界出力テンソルを含む合計サイズ (バイト)。

関連資料

[dpuGetInputTotalSize\(\)](#)

インクルード ファイル

n2cube.h

入力可能なインターフェイス

Vitis AI v1.0

dpuGetBoundaryIOTensor()

例

```
DPUTensor * dpuGetInputTotalSize  
(  
    DPUTask *task,  
    Const char  
        *tensorName  
)
```

引数

- task: DPU タスクへのポインター。
- tensorName: モデルがコンパイルされた後に VAI_C コンパイラによって一覧表示されるテンソル名。

説明

指定したテンソル名から DPU タスクの境界入力または出力テンソルを取得します。モデルがコンパイルされた後に VAI_C コンパイラによってテンソル名が一覧表示されます。

戻り値

DPUTensor へのポインター。

インクルード ファイル

n2cube.h

入力可能なインターフェイス

Vitis AI v1.0

dpuBindInputTensorBaseAddress()

書式

```
int dpuBindInputTensorBaseAddress
(
    DPUTask *task,
    int8_t *addrVirt,
    int8_t *addrPhy
)
```

引数

- task: DPU タスクへのポインター。
- addrVirt:
- addrPhy: DPU タスクのすべての境界出力テンソルを保持する DPU 出力メモリ バッファーの物理的アドレス。DPU タスクのすべての境界出力テンソルを保持する DPU 出力メモリ バッファーの仮想アドレス。

説明

入力メモリ バッファーの指定されたベース物理アドレスと仮想アドレスを DPU タスクに結び付ける。

注記: スプリット I/O モードで VAI_C によってコンパイルされた DPU カーネルに対してのみ使用可能。

戻り値

正常に完了すると 0 が返され、それ以外の場合はエラーが報告されます。

関連 API

[dpuBindOutputTensorBaseAddress\(\)](#)

インクルード ファイル

n2cube.h

対応バージョン

Vitis AI v1.0

dpuBindOutputTensorBaseAddress()

書式

```
int dpuBindOutputTensorBaseAddress
(
    DPUTask *task,
    int8_t *addrVirt,
    int8_t *addrPhy
)
```

引数

- task: DPU タスクへのポインター。
- addrVirt: DPU タスクのすべての境界出力テンソルを保持する DPU 出力メモリ バッファーの仮想アドレス。
- addrPhy: DPU タスクのすべての境界出力テンソルを保持する DPU 出力メモリ バッファーの物理的アドレス。

説明

出力メモリ バッファーの指定された物理ベース アドレスと仮想ベース アドレスを DPU タスクに結び付けます。

注記: スプリット I/O モードで VAI_C によってコンパイルされた DPU カーネルに対してのみ使用可能。

戻り値

正常に完了すると 0 が返され、それ以外の場合はエラーが報告されます。

関連 API

[dpuBindInputTensorBaseAddress\(\)](#)

インクルード ファイル

n2cube.h

対応バージョン

Vitis AI v1.0

Python API

モジュール n2cube のほとんどの Vitis AI の高度な低レベル Python API は、ライブラリ libn2cube の C++ API と同じです。それらの違いを次にリストします。詳細は以降のセクションで説明します。

- dpuGetOutputTensorAddress(): C++ API とは異なる戻り値の種類。
- dpuGetTensorAddress(): C++ API とは異なる戻り値の種類。
- dpuGetInputTensorAddress(): Python API には使用できない。
- dpuGetTensorData(): Python API にのみ使用可能。
- dpuGetOutputTensorInCHWInt8(): C++ API とは異なる戻り値の種類。
- dpuGetOutputTensorInCHWFP32(): C++ API とは異なる戻り値の種類。
- dpuGetOutputTensorInHWCInt8: C++ API とは異なる戻り値の種類。
- dpuGetOutputTensorInHWCFP32(): C++ API とは異なる戻り値の種類。
- dpuRunSoftmax(): C++ API とは異なる戻り値の種類。

さらに、Python インターフェイスには DPU スプリット IO を使用できません。したがって、Python を使用してモデルを運用する場合、次の 2 つの API は使用できません。

- `dpuBindInputTensorBaseAddress()`
- `dpuBindOutputTensorBaseAddress()`

API のリスト

モジュール `n2cube` の変更された Python API のプロトタイプとパラメーターについては、次のセクションで詳しく説明します。

`dpuGetOutputTensorAddress()`

例

```
dpuGetOutputTensorAddress
(
    task,
    nodeName,
    idx = 0
)
```

引数

- `task`: DPU タスクへの `ctypes` 型ポインター。
- `nodeName`: 文字列の DPU ノードの名前。

注記: DPU カーネルやタスクの入力ノード名は、ニューラル ネットワークが VAI_C によってコンパイルされた後にリストされます。無効なノード名が指定された場合、エラー メッセージが報告されます。

- `idx`: ノードの单一入力テンソルのインデックス。デフォルト値は 0。

説明

DPU タスクの出力テンソルのデータを指す `ctypes` 型ポインターを取得します。

注記: C++ API の場合は、DPU タスクの出力テンソルの `int8_t` 型の開始アドレスを返します。

戻り値

DPU タスクの出力テンソルのデータを指す `ctypes` 型ポインターを返します。`dpuGetTensorData` と組み合わせて使用すると、出力テンソルのデータを取得できます。

関連資料

[dpuGetTensorData\(\)](#)

インクルード ファイル

`n2cube`

入力可能なインターフェイス

Vitis AI v1.0

dpuGetTensorAddress()

例

```
dpuGetTensorAddress  
(  
    tensor  
)
```

引数

- tensor: DPU テンソルへの ctypes 型ポインター

説明

DPU タスクの出力テンソルのデータを指す ctypes 型ポインターを取得します。

注記: C++ API の場合は、DPU タスクの出力テンソルの int8_t 型の開始アドレスを返します。

戻り値

DPU テンソルのデータを指す ctypes 型ポインターを返します。dpuGetTensorData と組み合わせて使用すると、テンソルのデータを取得できます。

関連資料

[dpuGetTensorData\(\)](#)

インクルード ファイル

n2cube

入力可能なインターフェイス

Vitis AI v1.0

dpuGetTensorData()

例

```
dpuGetTensorData  
(  
    tensorAddress,  
    data,  
    tensorSize  
)
```

引数

- tensorAddress: DPU テンソルのデータへの ctypes 型ポインター。
- data: DPU テンソルのデータ。
- tensorSize: DPU テンソルのデータ サイズ。

説明

DPU テンソルのデータを取得する。

戻り値

なし

関連資料

[dpuGetOutputTensorAddress\(\)](#)

インクルード ファイル

n2cube

入力可能なインターフェイス

Vitis AI v1.0

dpuGetOutputTensorInCHWInt8()

例

```
dpuGetOutputTensorInCHWInt8
(
    task,
    nodeName,
    int size,
    idx = 0
)
```

引数

- task: DPU タスクへの ctypes 型ポインター。
- size: 文字列の DPU ノードの名前。
- idx: ノードの单一出力テンソルのインデックス。デフォルト値は 0。

説明

DPU タスクの出力テンソルを取得し、その INT8 型データを CHW (Channel*Height*Width) の順序で CPU メモリ バッファーに格納します。

戻り値

出力データを保持する NumPy 配列。エラーの場合、サイズは 0 になります。

関連資料

[dpuGetOutputTensorInCHWFP32\(\)](#)

[dpuGetOutputTensorInHWCInt8\(\)](#)

[dpuGetOutputTensorInHWCFP32\(\)](#)

インクルード ファイル

n2cube

入力可能なインターフェイス

Vitis AI v1.0

dpuGetOutputTensorInCHWFP32()

例

```
dpuGetOutputTensorInCHWFP32
(
task,
nodeName,
int size,
idx = 0
)
```

引数

- task: DPU タスクへの ctypes 型ポインター。
- nodeName: 文字列の DPU ノードの名前。
- size: 格納される出力データのサイズ (バイト単位)。
- idx: ノードの单一出力テンソルのインデックス。デフォルト値は 0。

説明

DPU タスクの出力テンソルデータを INT8 から float32 へ変換し、CHW (Channel*Height*Width) の順序で CPU メモリ バッファーに格納します。

戻り値

出力データを保持する NumPy 配列。エラーの場合、サイズは 0 になります。

関連資料

[dpuGetOutputTensorInCHWInt8\(\)](#)

[dpuGetOutputTensorInHWCIint8\(\)](#)

[dpuGetOutputTensorInHWCFP32\(\)](#)

インクルード ファイル

n2cube

入力可能なインターフェイス

Vitis AI v1.0

dpuGetOutputTensorInHWCInt8()

例

```
dpuGetOutputTensorInHWCInt8
(
task,
nodeName,
int size,
idx = 0
)
```

引数

- task: DPU タスクへの ctypes 型ポインター。
- nodeName: 文字列の DPU ノードの名前。
- size: 格納される出力データのサイズ (バイト単位)。
- idx: ノードの单一出力テンソルのインデックス。デフォルト値は 0。

説明

DPU タスクの出力テンソルを取得し、その INT8 型データを HWC (Height*Width*Channel) の順序で CPU メモリ バッファーに格納します。

戻り値

出力データを保持する NumPy 配列。エラーの場合、サイズは 0 になります。

関連資料

[dpuGetOutputTensorInCHWInt8\(\)](#)

[dpuGetOutputTensorInCHWF32\(\)](#)

[dpuGetOutputTensorInHWCFP32\(\)](#)

インクルード ファイル

n2cube

入力可能なインターフェイス

Vitis AI v1.0

dpuGetOutputTensorInHWCFP32()

例

```
dpuGetOutputTensorInHWCFP32
(
task,
nodeName,
int size,
idx = 0
)
```

引数

- task: DPU タスクへの ctypes 型ポインター。
- nodeName: 文字列の DPU ノードの名前。
- size: 格納される出力データのサイズ (バイト単位)。
- idx: ノードの单一出力テンソルのインデックス。デフォルト値は 0。

説明

DPU タスクの出力テンソルデータを INT8 から float32 へ変換し、HWC (Height*Width*Channel) の順序で CPU メモリ バッファーに格納します。

戻り値

出力データを保持する NumPy 配列。エラーの場合、サイズは 0 になります。

関連資料

[dpuGetOutputTensorInCHWInt8\(\)](#)

[dpuGetOutputTensorInCHWF32\(\)](#)

[dpuGetOutputTensorInHWCInt8\(\)](#)

インクルード ファイル

n2cube

入力可能なインターフェイス

Vitis AI v1.0

dpuRunSoftmax()

例

```
dpuRunSoftmax
(
    int8_t *input,
    int numClasses,
    int batchSize,
    float scale
)
```

引数

- input: int8_t 型の softmax 入力エレメントを格納するためのポインター。
- numClasses: softmax 計算が適用されるクラス数。
- batchSize: softmax 計算のバッチ サイズ。このパラメーターは、numClasses の入力値でエレメント数を分割した値を指定する必要があります。
- scale: softmax 計算の前に入力エレメントに適用されるスケール値。このパラメーターは、通常、API dpuGetTensorScale() を使用して取得できる。

説明

入力エレメントに対して softmax 計算を実行し、結果を出力メモリ バッファーに保存します。この API は、ハードウェア実装のモジュールを利用できる場合には、高速化のために DPU コアを活用します。「dexplorer -w」を実行して、DPU シグネチャ情報を表示します。

戻り値

softmax 計算の結果を保持する NumPy 配列。エラーの場合、サイズは 0 になります。

インクルード ファイル

n2cube.h

入力可能なインターフェイス

Vitis AI v1.0

その他のリソースおよび法的通知

ザイリンクス リソース

アンサー、資料、ダウンロード、フォーラムなどのサポート リソースは、[ザイリンクス サポート](#) サイトを参照してください。

Documentation Navigator およびデザイン ハブ

ザイリンクス Documentation Navigator (DocNav) では、ザイリンクスの資料、ビデオ、サポート リソースにアクセスでき、特定の情報を取得するためにフィルター機能や検索機能を利用できます。DocNav を開くには、次のいずれかを実行します。

- Vivado® IDE で [Help]→[Documentation and Tutorials] をクリックします。
- Windows で [スタート]→[すべてのプログラム]→[Xilinx Design Tools]→[DocNav] をクリックします。
- Linux コマンド プロンプトに「`docnav`」と入力します。

ザイリンクス デザイン ハブには、資料やビデオへのリンクがデザイン タスクおよびトピックごとにまとめられており、これらを参照することでキー コンセプトを学び、よくある質問 (FAQ) を参考に問題を解決できます。デザイン ハブにアクセスするには、次のいずれかを実行します。

- DocNav で [Design Hub View] タブをクリックします。
- ザイリンクス ウェブサイトで[デザイン ハブ](#) ページを参照します。

注記: DocNav の詳細は、ザイリンクス ウェブサイトの [Documentation Navigator](#) ページを参照してください。DocNav からは、日本語版は参照できません。ウェブサイトのデザイン ハブ ページをご利用ください。

参考資料

次の文書は、このユーザー ガイドの補足資料として役立ちます。日本語版のバージョンは、英語版より古い場合があります。

1. リリース ノートおよび既知の問題 - <https://github.com/Xilinx/Vitis-AI/blob/master/doc/release-notes/1.x.md>

お読みください: 重要な法的通知

本通知に基づいて貴殿または貴社(本通知の被通知者が個人の場合には「貴殿」、法人その他の団体の場合には「貴社」。以下同じ)に開示される情報(以下「本情報」といいます)は、ザイリンクスの製品を選択および使用することのためにのみ提供されます。適用される法律が許容する最大限の範囲で、(1)本情報は「現状有姿」、およびすべて受領者の責任で(with all faults)という状態で提供され、ザイリンクスは、本通知をもって、明示、默示、法定を問わず(商品性、非侵害、特定目的適合性の保証を含みますがこれらに限られません)、すべての保証および条件を負わない(否認する)ものとします。また、(2)ザイリンクスは、本情報(貴殿または貴社による本情報の使用を含む)に関係し、起因し、関連する、いかなる種類・性質の損失または損害についても、責任を負わない(契約上、不法行為上(過失の場合を含む)、その他のいかなる責任の法理によるかを問わない)ものとし、当該損失または損害には、直接、間接、特別、付随的、結果的な損失または損害(第三者が起こした行為の結果被った、データ、利益、業務上の信用の損失、その他あらゆる種類の損失や損害を含みます)が含まれるものとし、それは、たとえ当該損害や損失が合理的に予見可能であったり、ザイリンクスがそれらの可能性について助言を受けていた場合であったとしても同様です。ザイリンクスは、本情報に含まれるいかなる誤りも訂正する義務を負わず、本情報または製品仕様のアップデートを貴殿または貴社に知らせる義務も負いません。事前の書面による同意のない限り、貴殿または貴社は本情報を再生産、変更、頒布、または公に展示してはなりません。一定の製品は、ザイリンクスの限定的保証の諸条件に従うこととなるので、<https://japan.xilinx.com/legal.htm#tos>で見られるザイリンクスの販売条件を参照してください。IPコアは、ザイリンクスが貴殿または貴社に付与したライセンスに含まれる保証と補助的条件に従うことになります。ザイリンクスの製品は、フェイルセーフとして、または、フェイルセーフの動作を要求するアプリケーションに使用するために、設計されたり意図されたりしていません。そのような重大なアプリケーションにザイリンクスの製品を使用する場合のリスクと責任は、貴殿または貴社が単独で負うものです。<https://japan.xilinx.com/legal.htm#tos>で見られるザイリンクスの販売条件を参照してください。

自動車用のアプリケーションの免責条項

オートモーティブ製品(製品番号に「XA」が含まれる)は、ISO 26262 自動車用機能安全規格に従った安全コンセプトまたは余剰性の機能(「セーフティ設計」)がない限り、エアバッグの展開における使用または車両の制御に影響するアプリケーション(「セーフティ アプリケーション」)における使用は保証されていません。顧客は、製品を組み込むすべてのシステムについて、その使用前または提供前に安全を目的として十分なテストを行うものとします。セーフティ設計なしにセーフティ アプリケーションで製品を使用するリスクはすべて顧客が負い、製品の責任の制限を規定する適用法令および規則にのみ従うものとします。

著作権

© Copyright 2019-2021 Xilinx, Inc. Xilinx、Xilinx のロゴ、Alveo、Artix、Kintex、Spartan、Versal、Virtex、Vivado、Zynq、およびこの文書に含まれるその他の指定されたブランドは、米国およびその他各国のザイリンクス社の商標です。AMBA、AMBA Designer、Arm、ARM1176JZ-S、CoreSight、Cortex、PrimeCell、Mali、およびMPCoreは、EUおよびその他各国のArm Limitedの商標です。PCI、PCIe、およびPCI ExpressはPCI-SIGの商標であり、ライセンスに基づいて使用されています。

この資料に関するフィードバックおよびリンクなどの問題につきましては、jpn_trans_feedback@xilinx.comまで、または各ページの右下にある[フィードバック送信]ボタンをクリックすると表示されるフォームからお知らせください。フィードバックは日本語で入力可能です。いただきましたご意見を参考に早急に対応させていただきます。なお、このメールアドレスへのお問い合わせは受け付けておりません。あらかじめご了承ください。