第 4 世代インテル® Xeon® スケーラブル・プロセッサー上での AI のチューニング・ガイド

この記事は、インテル® デベロッパー・ゾーンに公開されている「Tuning Guide for AI on the 4th Generation Intel® Xeon® Scalable Processors」の日本語参考訳です。原文は更新される可能性があります。原文と翻訳文の内容が異なる場合は原文を優先してください。

原文更新日: 2023 年 2 月 1 日

はじめに

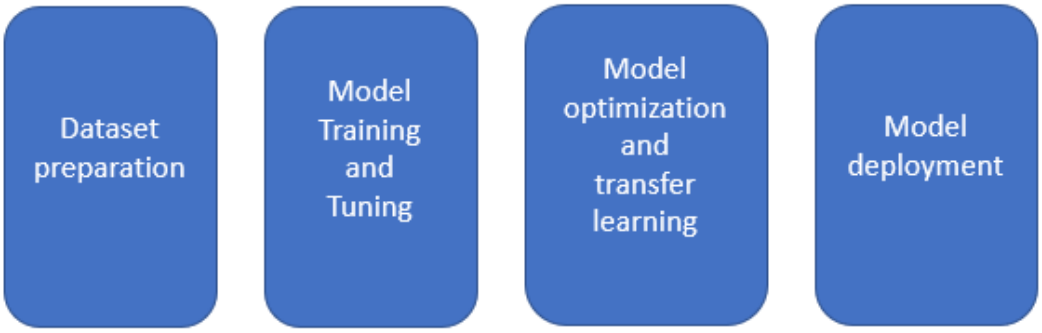
本記事は、インテル® AI アナリティクス・ツールキットとインテル® ディストリビューションの OpenVINO™ を使用しているユーザー向けです。インテルの最適化された AI ツールキットを使用して第 4 世代インテル® Xeon® スケーラブル・プロセッサー・プラットフォーム向けにチューニングを行う推奨事項を提供します。これらのハードウェアとソフトウェアの構成は、ほとんどの状況において最高のパフォーマンスを実現します。ただし、これらのツールの活用方法はさまざまであり、特定のシナリオに合わせてこれらの設定を慎重に検討する必要があります。

第 4 世代インテル® Xeon® スケーラブル・プロセッサー・プラットフォームは、HPC、AI、ビッグデータ、ネットワークなど、さまざまなワークロードを高速化し、パフォーマンスと TCO 効率を高めるように最適化された、ユニークで拡張性のあるプラットフォームです。

- 1 ソケットあたり最大 56 コア、8 ソケット・プラットフォームでは最大 448 コアを搭載。
- 新しい内蔵 AI アクセラレーション・エンジンのインテル® アドバンスト・マトリクス・エクステンション (インテル® AMX) により、BF16 および INT8 データ型をサポートし、さまざまな AI 推論およびトレーニングのワークロード (NLP、推薦システム、画像認識… など) を高速化。
- DDR5 と高帯域幅メモリー (HBM) によるメモリー帯域幅と速度の向上により、メモリー依存のワークロードに対応。
- 新しい内蔵インメモリー・データベース・アクセラレーター (IAX) によりデータ・アナリティクスまでの時間を短縮。
- PCIe* 5.0 による最大 2 倍の I/O 帯域幅で、レイテンシーに敏感なワークロードに高いスループットを提供。
- インテル® Dynamic Load Balancer (インテル® DLB) によりシステム全体のパフォーマンスを向上 - ネットワーク・データを効率良く処理。
- インテル® クイックアシスト・テクノロジー (インテル® QAT) により暗号化およびデータ圧縮ワークロードを最大 4 倍高速化。

AI の段階

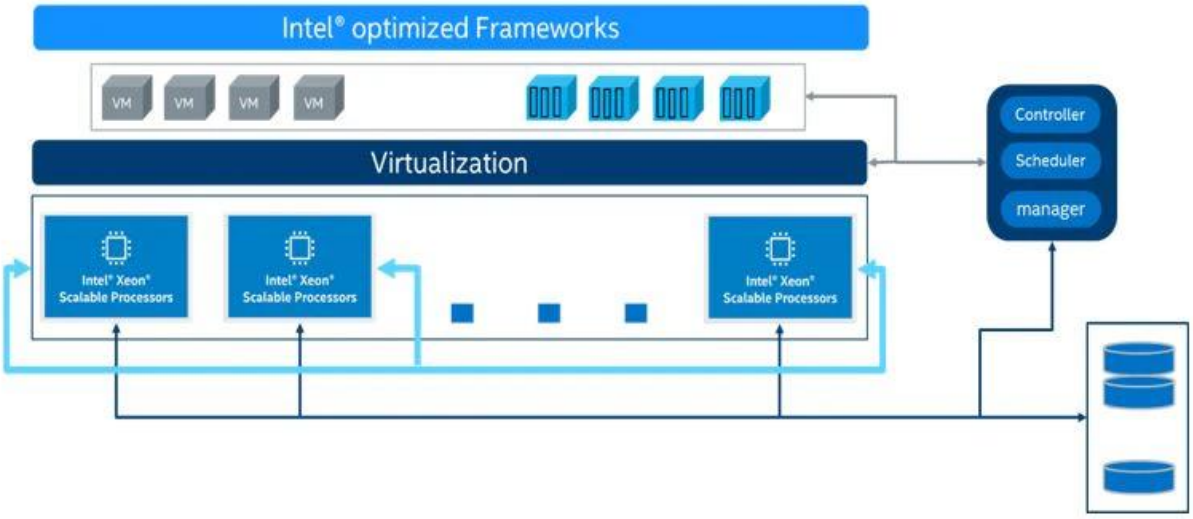
典型的なディープラーニング・アプリケーションには、以下の段階があります。



各段階では、以下のリソースの割り当てが必要になります。

- 計算能力
- メモリー
- データセットのストレージ
- 計算ノード間の通信リンク
- 最適化されたソフトウェア

適切なリソースの組み合わせを選択することで、AI サービスの効率が大幅に向上します。データセットの準備、モデルのトレーニング、モデルの最適化、モデルの展開など、すべてのプロセスは、トレーニングと推論を行うマシンラーニング/ディープラーニング・プラットフォームをサポートする、第 4 世代インテル® Xeon® スケーラブル・プロセッサー・プラットフォームに適用できます。以下に、提案するインフラストラクチャーを示します。



インテル® AMX

インテル® アドバンスト・ベクトル・エクステンション 512 (インテル® AVX-512) は、x86 プロセッサー・ベースの SIMD (Single Instruction, Multiple Data—単一命令複数データ) 命令セットです。1 つの命令で複数のデータ演算を実行できます。インテル® AVX-512 は、その名のとおり 512 ビットのレジスター幅を持ち、16 個の 32 ビット単精度浮動小数点数または 64 個の 8 ビット整数をサポートしています。

インテル® Xeon® スケーラブル・プロセッサーは、複雑な AI ワークロードを含むさまざまなワークロードをサポートし、インテル® ディープラーニング・ブースト (インテル® DL ブースト) を使用して AI 計算のパフォーマンスを向上します。インテル® DL ブーストには、AVX512_VNNI (ベクトル・ニューラル・ネットワーク命令)、AVX512_BF16、およびインテル® AMX が含まれています。

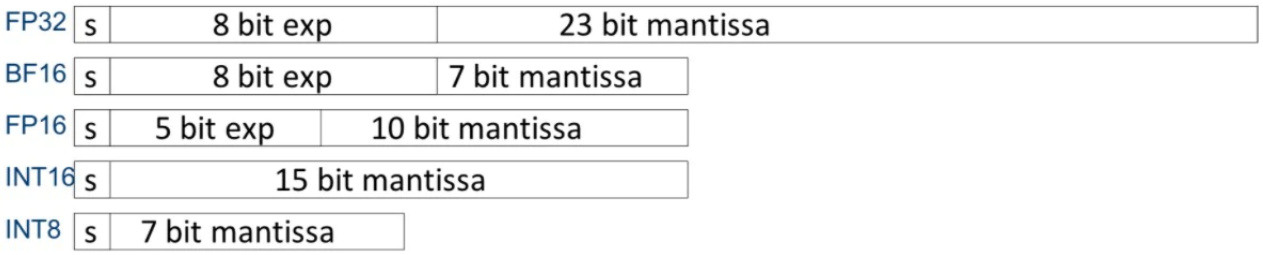
AVX512_VNNI は 3 つの命令 (`vpmaddubsw`、`vpmaddwd`、`vpaddd`) を 1 つ (`vpdpbusd`) の実行にまとめることができます。これにより、次世代インテル® Xeon® スケーラブル・プロセッサーが持つ演算能力をさらに引き出し、INT8 モデルの推論パフォーマンスを向上します。第 2 世代、第 3 世代、および第 4 世代のインテル® Xeon® スケーラブル・プロセッサーは、いずれも VNNI をサポートしています。

AVX-512_BF16 には、BF16 ペアのドット積を計算して単精度 (FP32) に集計する命令 (`VDPBF16PS`) と、パックド単精度データ (FP32) をパックド BF16 データに変換する命令 (`VCVTNE2PS2BF16`、`VCVTNEPS2BF16`) があります。

インテル® AMX は、より大きな 2 次元メモリーイメージからの部分配列を表す 2 次元レジスター (タイル) のセットと、タイルを操作できるアクセラレーターの 2 つのコンポーネントからなる新しい 64 ビット・プログラミング・パラダイムです。最初の実装は TMUL (Tile Matrix Multiply) ユニットと呼ばれています。

インテル® DL ブースト

プロセッサー	AVX512_VNNI	AVX512 BF16	インテル® AMX
Cascade Lake †	V		
Cooper Lake †	V	V	
Ice Lake †	V		
Sapphire Rapids †	V	V	V



S: 符号

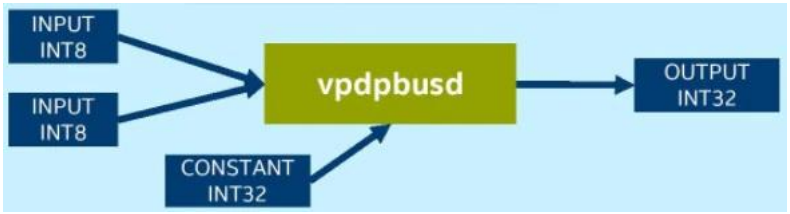
exp: 指数

mantissa: 仮数

ディープラーニング VNNI

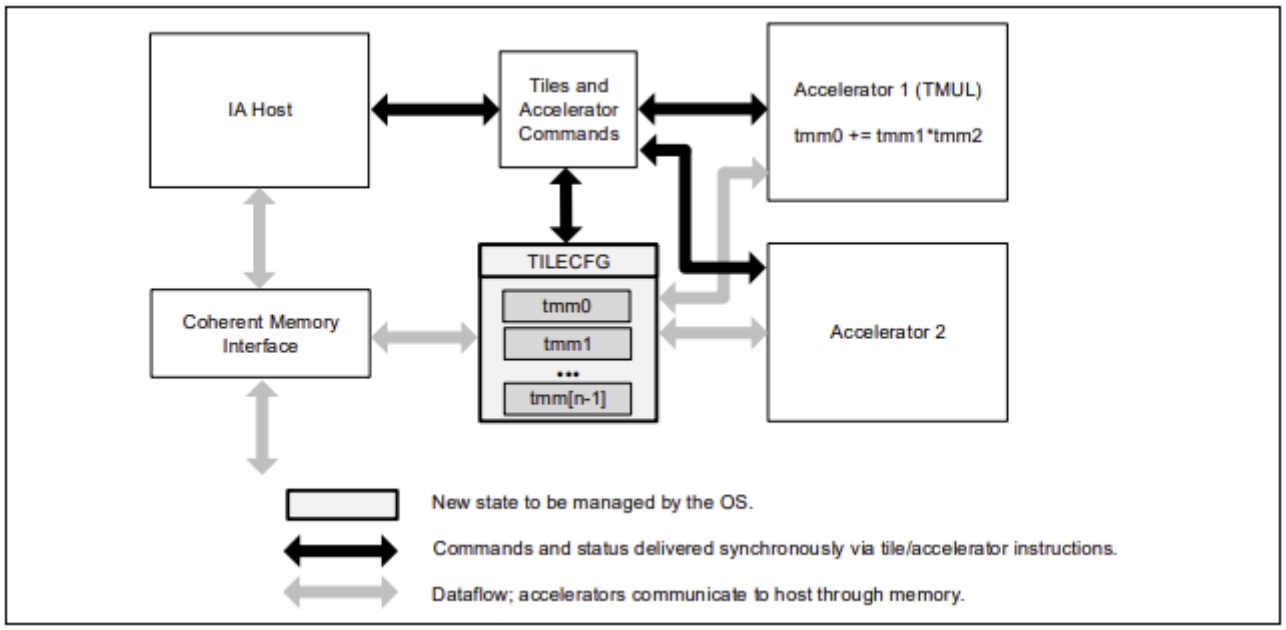
VNNI を使用しないプラットフォームでは、INT8 の畳み込み操作で乗累算を行うため、`vpmaddubsw`、`vpmaddwd`、および `vpaddd` 命令が必要です。

VNNI を使用するプラットフォームでは、INT8 の畳み込み操作には `vpdpbusd` 命令のみが必要です。



インテル® AMX

インテル® AMX では、「タイル」と呼ばれる 8 つのランク 2 テンソルレジスターを持つ新しいマトリクスレジスターを導入しています。また、これらのタイルを処理可能なアクセラレーターの概念も導入しています。マトリクス・レジスター・ファイルは、8 つのタイル (TMM0…TMM7) で構成され、それぞれの最大サイズは 16 行× 64 バイト列で、1 レジスターあたり 1KiB、レジスターファイル全体では 8KiB のサイズになります。大きな画像の小さな部分を表すタイルをメモリーにロードし、そのタイルを操作し、画像の次の部分を表す次のタイルでこれを繰り返します。終了したら、結果のタイルをメモリーに保存します。TMUL (Tile Matrix Multiply) ユニットは、タイルを操作できる FMA ユニットのグリッドで構成されています。



インテル® AMX サブエクステンションと関連命令

インテル® AMX は、AMX-TILE、AMX-INT8、および AMX-BF16 の 3 つのサブエクステンションから構成されています。

命令	AMX-TILE	AMX-INT8	AMX-BF16
LDTILECFG	V		
STTILECFG	V		
TILELOADD	V		
TILELOADDT1	V		
TILESTORED	V		
TILERELEASE	V		
TILEZERO	V		
TDPBSSD		V	
TDPBSUD		V	
TDPBUSD		V	
TDPBUUD		V	
TDPBF16PS			V

環境

本記事で説明するハードウェア構成は、第 4 世代インテル® Xeon® プロセッサーをベースにしています。サーバー・プラットフォーム、メモリー、ハードディスク、ネットワーク・インターフェイス・カードは、使用条件に応じて決定できます。本記事では、以下のハードウェアとソフトウェアを使用しました。

ハードウェア

ハードウェア	モデル
サーバー・プラットフォーム	インテル® サーバー・プラットフォーム開発コード名 Eagle Stream
CPU	インテル® Xeon® Platinum 8480 CPU @ 2.00GHz
メモリー	8x32GB DDR5、4800MT/秒

ソフトウェア

ソフトウェア	バージョン
オペレーティング・システム	Ubuntu* 22.04.1 LTS
カーネル	5.17.6
周波数ドライバー	intel_pstate
周波数ガバナー	performance

BIOS 設定

設定項目	推奨値
ソケット構成 → プロセッサー構成	
(新 BIOS) LP の有効化	シングルスレッド
(旧 BIOS) インテル® ハイパースレッディング・テクノロジー [すべて]	無効
ソケット構成 → 高度な電力管理設定	
CPU P ステート設定	以下の項目を「無効」に設定します。 Intel SpeedStep® テクノロジー (Pstates) インテル® ターボ・ブースト・テクノロジー CPU フレックス比のオーバーライド Perf P 制限
CPU C ステート制御	以下の項目を「無効」に設定します。 CPU C1 自動降格 CPU C1 自動降格解除 CPU C6 レポート、拡張停止状態 (C1E)
パッケージ C ステート制御 → パッケージ C ステート	C0/C1 ステート
ソケット構成 → 高度な電力管理設定 → 高度な PM チューニング → 電力性能バイアス調整	
電力性能チューニング	OS が EPB を制御

メモリー

すべての利用可能なメモリーチャネルを使用します。

CPU

第 4 世代インテル® Xeon® スケーラブル・プロセッサーは、全く新しいインテル® AMX 命令セットを搭載しています。インテル® AMX は、混合精度のディープラーニング・トレーニングおよび推論ワークロードを高速化します。第 4 世代インテル® Xeon® スケーラブル・プロセッサーの AMX_BF16 と AMX_INT8 という 2 つの命令セットは、それぞれ bfloat16 と int8 演算を高速化します。

注: CPU が AMX_BF16 と AMX_INT8 をサポートしているか確認するには、bash ターミナルで以下のコマンドを入力し、「flags」セクションでインテル® AMX を検索します。インテル® AMX 命令が見つからない場合は、Linux* カーネルを 5.17 以降に更新することを検討してください。

```
$ cat /proc/cpuinfo
```

ネットワーク

ノード間にまたがるトレーニング・クラスターが必要な場合、25Gbps/100Gbps などの高速ネットワークを選択するとスケーラビリティーが向上します。

ハードドライブ

I/O 効率を高めるには、SSD や読み書き速度の速いドライブを使用します。

Linux* オペレーティング・システムの最適化

処理を高速化するには、Linux* オペレーティング・システムを並列プログラミング向けに最適化します。

OpenMP* パラメーターの設定

OpenMP* (英語) は並列プログラミングの仕様です。OpenMP* ベースのスレッド化を実装しているアプリケーションでは、以下の環境変数を試し、最適な値を探してください。

- `OMP_NUM_THREADS` = コンテナ内の CPU コアの数
- `KMP_BLOCKTIME` = 1 または 0 (実際のモデルに応じて設定)
- `KMP_AFFINITY` = 粒度 (粒度には、`fine`、`verbose`、`compact`、`1`、`0` から指定)

CPU コア数

使用する CPU コアの数は、推論パフォーマンスに次のように影響します。

- バッチサイズが小さい場合 (オンラインサービスなど)、CPU コア数が増えると推論スループットの向上は徐々に小さくなります。使用するモデルにもよりますが、サービスの運用では 8～16 CPU コアが推奨されます。
- バッチサイズが大きい場合 (オフラインサービスなど)、CPU コア数が増えると推論スループットは直線的に増加する傾向があります。サービスの運用では 20 CPU コア以上が推奨されます。

```
$ taskset -C xxx-xxx -p pid (limits the number of CPU cores used in service)
```

NUMA 構成

NUMA ベースのサーバーでは、同一ノードで NUMA を構成すると、異なるノードで使用する場合と比較して、通常 5～10% パフォーマンスが向上します。

```
$ numactl -N NUMA_NODE -l command args ... (controls NUMA nodes running in service)
```

Linux* パフォーマンス・ガバナー

効率を重視します。最高のパフォーマンスを達成するため、CPU 周波数をピークに設定します。

```
$ cpupower frequency-set -g performance
```

CPU C ステートの設定

各 CPU には、C ステートまたは C モードと呼ばれるいくつかの電力モードが用意されています。CPU がアイドル状態のときに消費電力を抑えるため、CPU を低電力モードにすることができます。C ステートを無効にすることで、パフォーマンスが向上する可能性があります。

```
$ cpupower idle-set -d 2,3
```

TensorFlow* 向けインテル® オプティマイゼーションの使用

TensorFlow* 2.9 以降では、インテル® oneAPI ディープ・ニューラル・ネットワーク (インテル® oneDNN) ライブラリー (英語) によるパフォーマンスの向上がデフォルトで有効になります。TensorFlow* 向けインテル® オプティマイゼーションは、インテルのハードウェアで TensorFlow* に最新の機能と最適化を提供し、パフォーマンスを向上します。例えば、インテル® AVX-512 ベクトル・ニューラル・ネットワーク命令 (AVX512_VNNI) やインテル® AMX などです。

TensorFlow* 向けインテル® オプティマイゼーションは、オープンソース・プロジェクトとして https://github.com/Intel-tensorflow/tensorflow (英語) でリリースされています。

TensorFlow* 向けインテル® オプティマイゼーションのインストール

注: 第 4 世代インテル® Xeon® スケーラブル・プロセッサー向けの最適化は、TensorFlow* 向けインテル® オプティマイゼーション 2.11 から追加されており、利用には dev パッケージ (英語) が必要です。

```
$ conda create -n intel-tf python=3.8 -y
$ conda activate intel-tf
$ pip install intel-tensorflow==2.11.dev202242
```

TensorFlow* ベースのディープラーニング・モデルの最適化の推奨事項

インテル® プラットフォームでディープラーニングのパフォーマンスを最大限に引き出すため、TensorFlow* 向けインテル® オプティマイゼーションは以下の手法を利用しています。

- NUMA 制御: `numactl` は NUMA のスケジューリングとメモリー配置ポリシーを指定します。
- マルチスレッド: OpenMP* を利用して CPU コア間でディープラーニング・モデルの実行を並列化します。
- スレッド・アフィニティー: `KMP_AFFINITY` を使用して、特定のスレッドの実行をマルチプロセッサー・コンピューターの物理処理ユニットのサブセットに制限します。

詳細は、「CPU 上で TensorFlow* のパフォーマンスを最大化」(英語) を参照してください。

BF16 の有効化

第 4 世代インテル® Xeon® スケーラブル・プロセッサーは、インテル® DL ブーストとインテル® AMX により、BF16 や INT8 などの低精度データ型を使用した AI 推論の高速化をサポートします。AI モデルを高速化するため、AMX_BF16、AMX_INT8、AVX512_BF16、AVX512_VNNI などの命令が用意されています。

推論

事前にトレーニングされた FP32 モデルの場合 (ここでは TensorFlow* Hub の resnet50 を例として使用):

1. TensorFlow* 向けインテル® オプティマイゼーションに加え、このサンプル用に tensorflow_hub をインストールします。

   ```
   $ pip install tensorflow_hub
   ```

2. `export ONEDNN_VERBOSE=1` で推論を実行します。AVX512_BF16 命令と AMX_BF16 命令が有効であることを確認できるはずです。
   - 以下のコードを `inference.py` として保存します。

     ```
     import os
     import tensorflow as tf
     import tensorflow_hub as tf_hub

     # Enable Auto Mixed Precision
     tf.config.optimizer.set_experimental_options({"auto_mixed_precision_o
     nednn_bfloat16": True})

     os.environ["TFHUB_CACHE_DIR"] = 'tfhub_models'
     ```

```
model                                                                    =
tf_hub.KerasLayer('https://tfhub.dev/google/imagenet/resnet_v1_50/cla
ssification/5')
model(tf.random.uniform((1, 224, 224, 3)))
```

- ターミナルで推論スクリプトを実行します。

```
$ export ONEDNN_VERBOSE=1
$ python inference.py
```

- 結果を確認します。

```
dst_bf16::blocked:Adcb16a:f0,,,64x3x7x7,69.343
onednn_verbose,exec,cpu,convolution,jit:avx512_core_amx_bf16,forward_
training,src_bf16::blocked:acdb:f0 wei_bf16::blocked:Adcb16a:f0
bia_undef::undef::f0 dst_bf16::blocked:acdb:f0,attr-scratchpad:user
attr-post-
ops:binary_sub:f32:2+binary_mul:f32:2+binary_mul:f32:2+binary_add:f32
:2+eltwise_relu ,alg:convolution_direct,mb1_ic3oc64_ih224oh112kh7sh2d
h0ph3_iw224ow112kw7sw2dw0pw3,3.29614
onednn_verbose,exec,cpu,pooling_v2,jit:avx512_core_bf16,forward_train
ing,src_bf16::blocked:acdb:f0 dst_bf16::blocked:acdb:f0
ws_u8::blocked:acdb:f0,,alg:pooling_max,mb1ic64_ih112oh56kh3sh2dh0ph0
_iw112ow56kw3sw2dw0pw0,5.21802
```

トレーニング

TensorFlow* Keras API を使用して BF16 で混合精度モデルをトレーニングすることもできます。

- 以下のコードを `training.py` として保存します。

```
import tensorflow as tf
from keras.utils import np_utils
from tensorflow.keras import mixed_precision

# Enable Auto Mixed Precision
mixed_precision.set_global_policy('mixed_bfloat16')

# load data
cifar10 = tf.keras.datasets.cifar10
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
num_classes = 10

# pre-process
x_train, x_test = x_train/255.0, x_test/255.0
y_train = np_utils.to_categorical(y_train, num_classes)
y_test = np_utils.to_categorical(y_test, num_classes)

# build model
feature_extractor_layer = tf.keras.applications.ResNet50(include_top=False,
weights='imagenet')
feature_extractor_layer.trainable = False
model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(32, 32, 3)),
    feature_extractor_layer,
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(1024, activation='relu'),
    tf.keras.layers.Dropout(0.2),
```

```
    tf.keras.layers.Dense(num_classes, activation='softmax')
])
model.compile(
  optimizer=tf.keras.optimizers.Adam(),
  loss=tf.keras.losses.CategoricalCrossentropy(),
  metrics=['acc'])

# train model
model.fit(x_train, y_train,
    batch_size = 128,
    validation_data=(x_test, y_test),
    epochs=1)

model.save('resnet_bf16_model')
```

- ターミナルでトレーニング・スクリプトを実行します。

```
$ export ONEDNN_VERBOSE=1
$ python training.py
```

- 結果を確認します。

```
dst_bf16::blocked:Adcb16a:f0,,,64x3x7x7,160.375
onednn_verbose,exec,cpu,convolution,jit:avx512_core_amx_bf16,forward_traini
ng,src_bf16::blocked:acdb:f0 wei_bf16::blocked:Adcb16a:f0
bia_bf16::blocked:a:f0 dst_bf16::blocked:acdb:f0,attr-
scratchpad:user ,alg:convolution_direct,mb128_ic3oc64_ih32oh16kh7sh2dh0ph3_
iw32ow16kw7sw2dw0pw3,2.0481
onednn_verbose,exec,cpu,batch_normalization,bnorm_jit:avx512_core_bf16,forw
ard_inference,data_bf16::blocked:acdb:f0
diff_undef::undef::f0,,flags:GSR,mb128ic64ih16iw16,0.783203
onednn_verbose,exec,cpu,pooling_v2,jit:avx512_core_bf16,forward_training,sr
c_bf16::blocked:acdb:f0 dst_bf16::blocked:acdb:f0
ws_u8::blocked:acdb:f0,,alg:pooling_max,mb128ic64_ih18oh8kh3sh2dh0ph0_iw18o
w8kw3sw2dw0pw0,0.908936
```

INT8 の使用

TensorFlow* 向けインテル® オプティマイゼーションは、インテル® ニューラル・コンプレッサー (英語) と連携して、互換性のある TensorFlow* INT8 量子化ソリューションを同じユーザー体験で提供します。

PyTorch* 向けインテル® エクステンションによる最適化とパフォーマンスの向上

PyTorch* 向けインテル® エクステンションは、インテルのハードウェアで PyTorch* に最新の機能と最適化を提供し、パフォーマンスを向上します。ほとんどの最適化は、stock PyTorch* に含まれる予定です。例えば、インテル® AVX-512 ベクトル・ニューラル・ネットワーク命令 (AVX512_VNNI) やインテル® AMX などです。

PyTorch* 向けインテル® エクステンションは、オープンソース・プロジェクトとして https://github.com/intel/intel-extension-for-pytorch (英語) でリリースされています。

PyTorch* ディープラーニング・フレームワークの使用

拡張機能が正しく動作するように、PyTorch* がインストールされていることを確認してください。インストールの詳細は、https://intel.github.io/intel-extension-for-pytorch/latest/tutorials/installation.html# (英語) を参照してください。

PyTorch* の展開

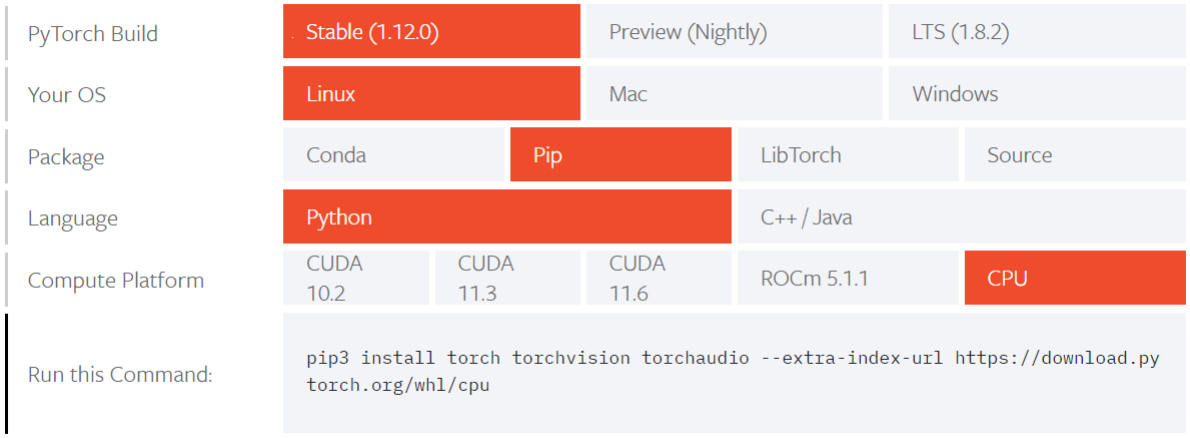
参考資料: https://intel.github.io/intel-extension-for-pytorch/latest/index.html (英語)

環境: Python* 3.7 以上

ステップ 1: PyTorch* の公式ウェブサイト https://pytorch.org/ (英語) にアクセスします。

ステップ 2: CPU を選択します。

現在、インテル® oneDNN は PyTorch* の公式バージョンに統合されているため、追加のインストールなしで、インテル® Xeon® スケーラブル・プロセッサー・プラットフォームで優れたパフォーマンスを発揮できます。**[Compute Platform]** に **[CPU]** を選択します。詳細は以下の図を参照してください。



ステップ 3: インストールします。

```
$ pip3 install torch torchvision torchaudio --extra-index-url
https://download.pytorch.org/whl/cpu
```

PyTorch* 向けインテル® エクステンションのインストール

以下のいずれかのコマンドを使用して、PyTorch* 向けインテル® エクステンションをインストールできます。

```
$ python -m pip install intel_extension_for_pytorch
$ python -m pip install intel_extension_for_pytorch -f
https://software.intel.com/ipex-whl-stable
```

実際に使ってみる

PyTorch* 向けインテル® エクステンションを使用するには、わずかなコード変更が必要です。PyTorch* 命令モードと TorchScript モードの両方がサポートされます。

推論では、`ipex.optimize` 関数をモデル・オブジェクトに適用します。

トレーニングでは、`ipex.optimize` 関数をモデル・オブジェクトとオプティマイザー・オブジェクトに適用します。

以下は、BF16/FP32 データ型を使用したトレーニング・コードの例です。その他のトレーニングと推論のサンプルは、サンプルページ (英語) を参照してください。

```
import torch
import intel_extension_for_pytorch as ipex

model = Model()
model = model.to(memory_format=torch.channels_last)
criterion = ...
optimizer = ...
model.train()

# For FP32
model, optimizer = ipex.optimize(model, optimizer=optimizer)

# For BF16
model, optimizer = ipex.optimize(model, optimizer=optimizer,
dtype=torch.bfloat16)

# Setting memory_format to torch.channels_last could improve performance with 4D
input data. This is optional.
data = data.to(memory_format=torch.channels_last)
optimizer.zero_grad()
output = model(data)
```

PyTorch* ベースのディープラーニング・モデルのトレーニングと推論における最適化の推奨事項

PyTorch* と PyTorch* 向けインテル® エクステンションのデフォルトのプリミティブは高度に最適化されていますが、さらにパフォーマンスを向上する追加の設定オプションがあります。その多くは、設定オプションを自動で設定する起動スクリプトによって適用でき、主に以下のようなものがあります。

1. OpenMP* ライブラリー: [インテルの OpenMP* ライブラリー (デフォルト) | GNU* OpenMP* ライブラリー]
2. メモリー・アロケーター: [PyTorch* のデフォルトのメモリー・アロケーター | Jemalloc | TCMalloc (デフォルト)]
3. インスタンス数: [1 インスタンス (デフォルト) | 複数のインスタンス]

詳細は、「起動スクリプトの使用ガイド」 (英語) を参照してください。

起動スクリプトのほかにも、インテル® CPU や NUMA (Non-Uniform Memory Access) 構成の設定など、ハードウェアの設定も可能です。ソフトウェア設定では、チャネル・ラスト・メモリー・フォーマット、OpenMP*、`numactl` を利用して CPU の計算リソースを最大限に活用し、PyTorch* 向けインテル® エクステンションによ

りパフォーマンスを向上するように設定できます。詳細は、「パフォーマンス・チューニング・ガイド」 (英語) を参照してください。

BF16 推論の使用

第 4 世代インテル® Xeon® スケーラブル・プロセッサーは、インテル® DL ブーストとインテル® AMX により、BF16 や INT8 などの低精度データ型を使用した AI 推論の高速化をサポートします。AI モデルを高速化するため、AMX_BF16、AMX_INT8、AVX512_BF16、AVX512_VNNI などの命令が用意されています。

自動混合精度 (AMP)

`torch.cpu.amp` は、実行時に自動でデータ型を変換します。ディープラーニングのワークロードは、計算負荷が小さく、メモリー使用量が少ない `torch.float16` や `torch.bfloat16` などの低精度浮動小数点データ型の恩恵を受けることができます。自動混合精度 (AMP) 機能は、すべての演算子のデータ型変換を自動化します。

AMX_BF16 を有効にする手順

CPU マシンが AMX_BF16 命令をサポートしているか確認するには、`lscpu` コマンドを使用します。

`torch.cpu.amp.autocast` は、スクリプトのスコープを混合精度で実行できるようにします。これらのスコープでは、精度を維持しながらパフォーマンスを向上するため、autocast クラスによって選択されたデータ型で演算が実行されます。次のような単純なネットワークでは、混合精度によるスピードアップが見られるはずです。

```
class SimpleNet(torch.nn.Module):
    def __init__(self):
        super(SimpleNet, self).__init__()
        self.conv = torch.nn.Conv2d(64, 128, (3, 3), stride=(2, 2), padding=(1,
1), bias=False)

    def forward(self, x):
        return self.conv(x)
```

`torch.cpu.amp.autocast` は、スクリプトのスコープを混合精度で実行できるようにするコンテキスト・マネージャーとして設計されています。AMX_BF16 は AVX512_BF16 より新しく、より高度な組込み関数です。AI アプリケーションをサポートする、優れたパフォーマンスを提供します。そのため、BFloat16 データ型では、AMX_BF16 の実行優先度が最も高くなります。インテルによって最適化された AI フレームワークは、まず AMX_BF16 を選択します。利用できない場合は、AVX512_BF16 が選択されます。詳細は、「自動混合精度 (AMP)」 (英語) を参照してください。

```
model = SimpleNet().eval()
    x = torch.rand(64, 64, 224, 224)
with torch.cpu.amp.autocast():
    y = model(x)
```

AMX_BF16 が有効かどうかを確認するには、`avx512_core_amx_bf16` JIT Kernel の使用状況を確認します。`ONEDNN_VERBOSE=1` に設定されていることを確認します。

PyTorch* 向けインテル® エクステンションの GitHub* リンクは、こちら (英語) です

AI ニューラル・ネットワーク・モデルの低精度最適化

概要

第 4 世代インテル® Xeon® スケーラブル・プロセッサーは、インテル® DL ブーストとインテル® AMX により、BF16 や INT8 などの低精度データ型を使用した AI 推論の高速化をサポートします。

1. 量子化

   INT8 データ型を使用した AI モデルの最適化は、量子化の一種であり、連続する無限値をより小さな離散有限値の集合にマッピングするプロセスです。

2. 混合精度

   混合精度は、BF16 などの低精度データ型を使用し、トレーニングや推論時に 16 ビットと 32 ビットの混合浮動小数点型で動作するモデルを作成します。これにより、より少ないメモリー消費で高速に実行されます。

命令

第 4 世代インテル® Xeon® スケーラブル・プロセッサーは AI モデルを高速化する命令をサポートしています。

組込み関数	AVX512_VNNI	AVX512_BF16	AMX_INT8	AMX_BF16
データ型	INT8	BF16	INT8	BF16
変数タイプ	ベクトル	ベクトル	行列	行列

AMX_INT8/AMX_BF16 は AVX512_VNNI/AVX512_BF16 より新しく、より高度な組込み関数です。AI アプリケーションをサポートする、優れたパフォーマンスを提供します。AMX_INT8 は実行優先度が最も高くなります。インテルによって最適化された AI フレームワークは、まず AMX_INT8 を選択します。利用できない場合は、AVX512_VNNI が選択されます。

手順

1. FP32 モデルを INT8/BF16 モデルに変換します。

   量子化処理または混合精度処理を実行し、INT8/BF16 モデルを取得します。

2. 第 4 世代インテル® Xeon® スケーラブル・プロセッサー上で INT8/BF16 モデルの推論を、インテル® アーキテクチャー向けに最適化された AI フレームワークで実行します。

   AI フレームワークは、高速に実行するため CPU でサポートされている最高レベルの組込み関数を呼び出します。

例えば、AI フレームワークが AMX_INT8 ではなく AVX512_VNNI を呼び出す場合、新しいリリースが AMX_INT8 をサポートしているかどうか確認し、適切なリリースをインストールしてください。

AI ニューラル・ネットワークの量子化処理

ニューラル・ネットワークの計算は、主に畳み込み層と全結合層に集約されます。この 2 つの層での計算は次のように表すことができます。

Y = X * 重み + バイアス

パフォーマンスを最適化するため、行列乗算に注目するのは自然なことです。ニューラル・ネットワーク・モデルの量子化は、パフォーマンスと精度 (限定的) のトレードオフですが、行列演算を 32 ビット浮動小数点数から低精度整数に置き換えることで、計算を高速化するだけでなく、モデルを圧縮し、メモリー帯域幅を節約できます。

ニューラル・ネットワーク・モデルの量子化には、3 つのアプローチがあります。

- PTQ (Post-Training Quantization): ほとんどの AI フレームワークでサポートされています。トレーニング済みの FP32 モデルを量子化します。キャリブレーション処理 (小さなデータセットで FP32 モデルを推論する) を行い、各層の入力と出力のデータ範囲を記録します。そして、そのデータ範囲の情報に従ってモデルを量子化します。これにより、良好な量子化結果が得られます。
- QAT (Quantization-Aware-Training): トレーニングが収束した時点で FP32 モデルに `FakeQuantization` ノードを挿入します。これは量子化によって発生するノイズを増加させます。トレーニングのバックプロパゲーション (誤差逆伝播) 段階で、モデルの重みが有限区間内になるため、量子化の精度が向上します。
- DQ (Dynamic Quantization): PTQ と非常によく似ています。どちらもトレーニング後のモデルに対して使用される量子化手法です。違いは、DQ では活性化層での量子化係数がニューラル・ネットワーク・モデルの実行時に使用するデータ範囲によって動的に決定されるのに対し、PTQ では小規模な前処理済みデータセットのサンプルを用いて活性化層でのデータ分布と範囲情報を取得し、新たに生成した量子化モデルに恒久的に記録します。後述するインテル® ニューラル・コンプレッサーの `onnxruntime` はこの方式をバックエンドのみでサポートしています。

ニューラル・ネットワークのトレーニング後の量子化の基本的な手順は、以下のとおりです。

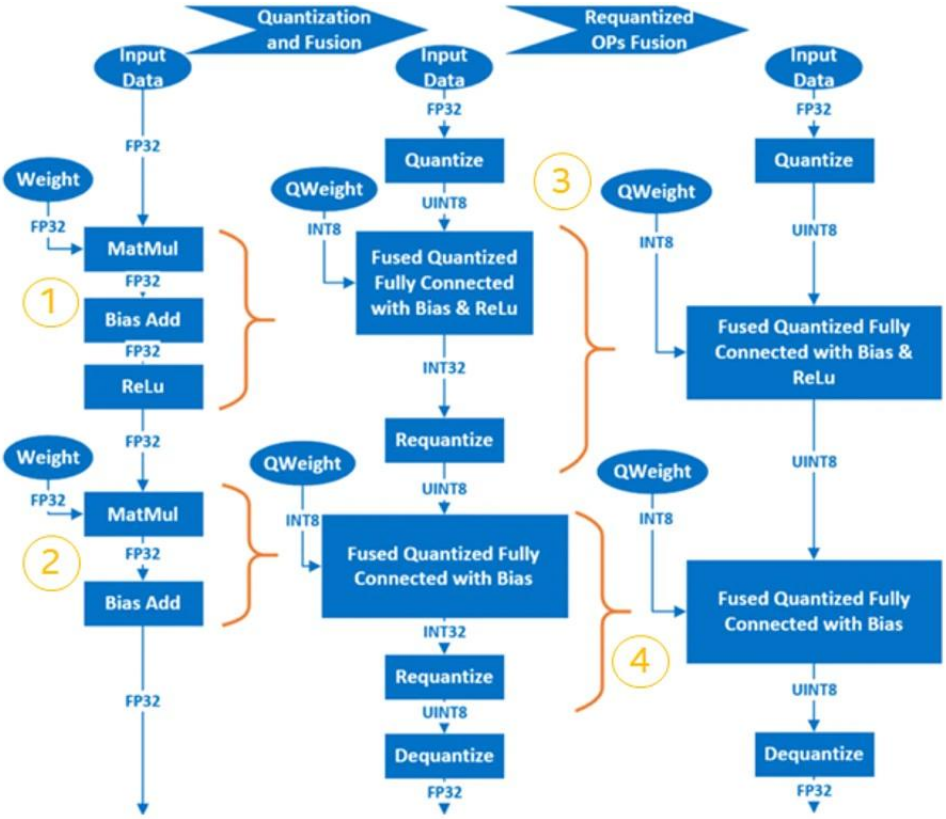
1. FP32 OP を INT8 OP に融合します。例えば、`MatMul`、`BiasAdd`、`ReLU` は、全結合層で 1 つの量子化 OP である `QuantizedMatMulWithBiasAndRelu` に融合できます。ニューラル・ネットワークのフレームワークによって、融合可能な OP は異なります。後述するインテル ® ニューラル・コンプレッサーの場合、TensorFlow* でサポートされる融合可能な OP の一覧は以下のページで確認できます。
   https://github.com/intel/neural-compressor/blob/master/neural_compressor/adaptor/tensorflow.yaml#L110 (英語)

   pyTorch* でサポートされる融合可能な OP の一覧は以下のページで確認できます。
   https://github.com/intel/neural-compressor/blob/master/neural_compressor/adaptor/pytorch_cpu.yaml#L251 (英語)

2. 重みを量子化し、量子化されたモデルに保存します。

3. キャリブレーション・データセットをサンプリングして入力/活性化層を量子化し、活性化層のデータの分布と範囲情報を取得し、新しく生成された量子化モデルに記録します。
4. 再量子化 OP は、対応する INT8 OP に融合され、最終的な量子化モデルが生成されます。

例えば、2 層の `MatMul` を含む簡単なモデルでは、量子化プロセスは以下のようになります。



AI ニューラル・ネットワークの混合精度処理

FP32 と BF16 は、データ範囲が同じで小数点以下の桁数が異なります。FP32 から BF16 に変換するのは簡単です。しかし、BF16 を使用すると、小数点以下が切り捨てられるため、モデルの精度に影響します。演算によっては、BF16 でも数値的に安全なものがあり、それらは FP32 と比較して明らかに精度が低下することはありません。一方、BF16 では数値的に危険な演算は、FP32 と比較して明らかに精度が低下します。以下の手順は、AI フレームワーク/ツールによって自動的に処理されます。数値的に危険な演算を許可するか、拒否するかを設定できます。

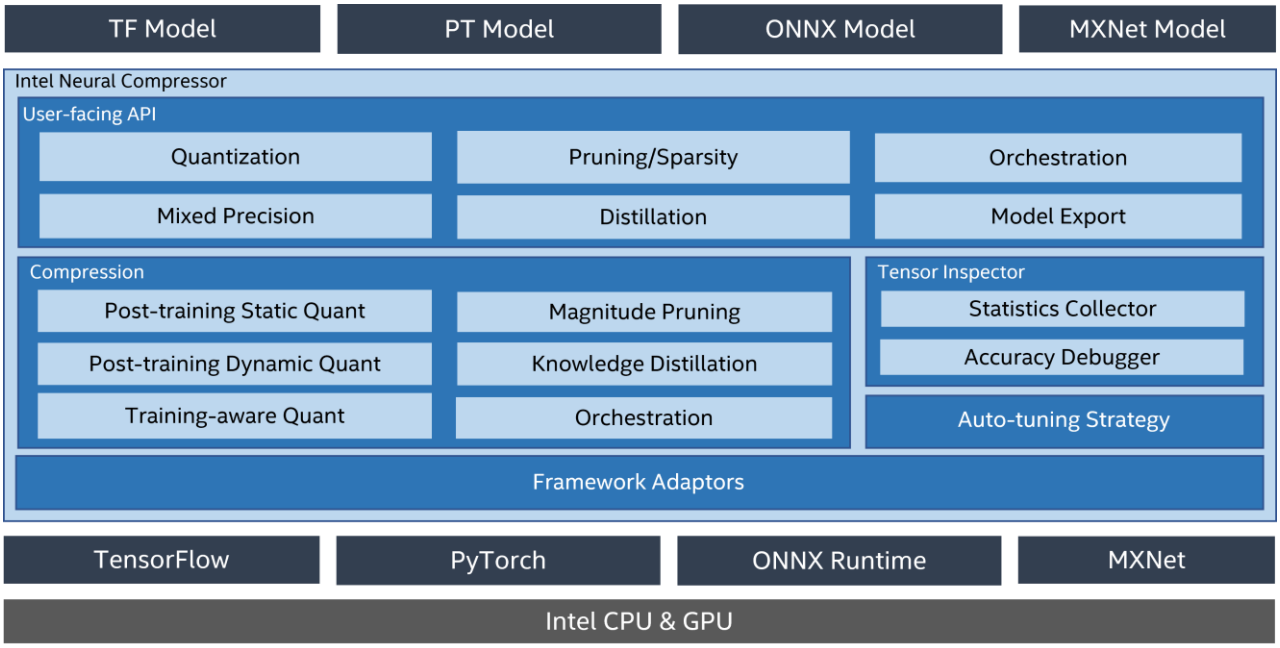
1. 数値的に安全な演算を許可するか、数値的に危険な演算を拒否するかを設定します。
2. 前のステップで許可した演算を BF16 に変換します。
3. FP32 と BF16 の操作の間に Cast 操作を挿入して、データを FP32 から BF16 に変換したり、BF16 から FP32 に戻します。

インテル® ニューラル・コンプレッサー

インテル® ニューラル・コンプレッサー (英語) は、インテル® oneAPI AI アナリティクス・ツールキットの主要な AI ソフトウェア・コンポーネントの 1 つで、インテルの CPU と GPU 上で動作するオープンソースの Python*

ライブラリーです。このツールキットは、量子化、プルーニング、知識蒸留などの一般的なネットワーク圧縮技術に対して、複数のディープラーニング・フレームワークにわたる統一されたインターフェイスを提供します。このツールは、ユーザーが最適な量子化モデルを素早く見つけられるように、精度を意識した自動チューニングをサポートしています。また、事前に定義された目標を使用してプルーニングしたモデルを生成するため、異なるウェイト・プルーニング・アルゴリズムを実装し、教師モデルから生徒モデルへの知識の蒸留を支援します。

参考資料: https://github.com/intel/neural-compressor (英語)



インテル® ニューラル・コンプレッサーは、現在、インテルにより最適化された以下のディープラーニング・フレームワークをサポートしています。

- Tensorflow* (英語)
- PyTorch* (英語)
- Apache* MXNet (英語)
- ONNX* ランタイム (英語)

検証済みのフレームワークとそのバージョンを以下に示します。

- OS バージョン: CentOS* 8.4、Ubuntu* 20.04
- Python* バージョン: 3.7、3.8、3.9、3.10

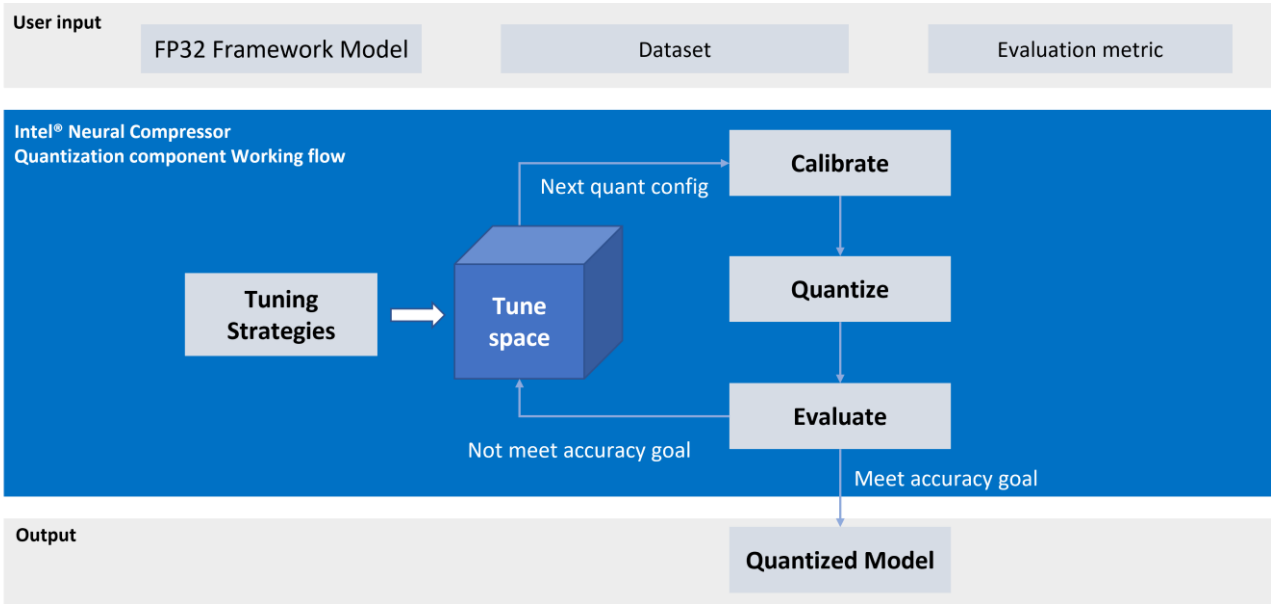
フレームワーク	TensorFlow*	TensorFlow* 向けインテル® オプティマイゼーション	PyTorch*	PyTorch* 向けインテル® エクステンション	ONNX* ランタイム	MXNet
バージョン (リンク先はすべて英語)	2.9.1 2.8.2 2.7.3	2.9.1 2.8.0 2.7.0	1.12.0+cpu 1.11.0+cpu 1.10.0+cpu	1.12.0 1.11.0 1.10.0	1.11.0 1.10.0 1.9.0	1.8.0 1.7.0 1.6.0

注: TensorFlow* 2.6～2.8 では、環境変数 `TF_ENABLE_ONEDNN_OPTS=1` を設定して、インテル® oneDNN による最適化を有効にしてください。TensorFlow* 2.9 以降では、インテル® oneDNN による最適化はデフォルトで有効になります。

インテル® ニューラル・コンプレッサーは以下のチューニング手法をサポートしています。

- Basic (英語)
- Bayesian (英語)
- MSE (英語)
- TPE (英語)
- Exhaustive (英語)
- Random (英語)

インテル® ニューラル・コンプレッサーのワークフローを以下に示します。設定されたチューニング手法に応じて、精度損失目標に合致するモデル量子化パラメーターが自動的に選択され、量子化モデルが生成されます。



インテル® ニューラル・コンプレッサーのインストール

インストールに関する詳細は、https://github.com/intel/neural-compressor#installation (英語) を参照してください。

ステップ 1: Anaconda* を使用して、`env_inc` という名前で Python* 3.x の仮想環境を作成します。この例では、Python* 3.9 を使用しています。

```
$ conda create -n env_inc python=3.9
$ conda activate env_inc
```

ステップ 2: バイナリーファイルを使用してインテル® ニューラル・コンプレッサーをインストールします。

```
# install stable basic version from pip
$ pip install neural-compressor
# install stable full version from pip (including GUI)
$ pip install neural-compressor-full
```

または

```
# install nightly basic version from pip
$ pip install -i https://test.pypi.org/simple/ neural-compressor
# install nightly full version from pip (including GUI)
$ pip install -i https://test.pypi.org/simple/ neural-compressor-full
```

または

```
# install stable basic version from from conda
$ conda install neural-compressor -c conda-forge -c intel
# install stable full version from from conda (including GUI)
$ conda install neural-compressor-full -c conda-forge -c intel
```

ステップ 3: AI フレームワークをインストールします。

扱うモデルの種類に応じて、Tensorflow*、PyTorch*、ONNX-RT*、MXNet をインストールします。例えば、intel-tensorflow をインストールする場合は以下のコマンドを実行します。

```
# install intel-tensorflow from pip
$ pip install intel-tensorflow
```

インテル® ニューラル・コンプレッサーの使用

ここでは、ResNet50 v1.0 を例に量子化と混合精度の最適化を行う方法を説明します。

データセットの準備

ステップ 1: ImageNet 検証用データセットをダウンロードして展開します。

生画像をダウンロードするには、image-net.org にアカウントを作成する必要があります。

```
$ mkdir -p img_raw/val
$ cd img_raw
$ wget http://www.image-
net.org/challenges/LSVRC/2012/xxxxxxxx/ILSVRC2012_img_val.tar
$ tar -xvf ILSVRC2012_img_val.tar -C val
```

ステップ 2: 画像ファイルをラベル順にサブ・ディレクトリーに移動します。

```
$ cd val
$ wget -qO-
https://raw.githubusercontent.com/soumith/imagenetloader.torch/master/valprep.sh
| bash
```

ステップ 3: prepare_dataset.sh (英語) スクリプトを使用して生データを 128 シャードの TFrecord 形式に変換します。

```
$ git clone https://github.com/intel/neural-compressor.git
$ cd neural-compressor/
$ git checkout 6663f7b
$ cd examples/tensorflow/image_recognition/tensorflow_models/quantization/ptq
bash prepare_dataset.sh --output_dir=./data --raw_dir=/PATH/TO/img_raw/val/ --
shards=128 --subset=validation
```

参考資料: https://github.com/intel/neural-compressor/tree/master/examples/tensorflow/image_recognition/tensorflow_models/quantization/ptq#2-prepare-dataset (英語)

事前トレーニング済み FP32 モデルの準備

```
$ wget https://storage.googleapis.com/intel-optimized-
tensorflow/models/v1_6/resnet50_fp32_pretrained_model.pb
```

量子化の実行

examples/tensorflow/image_recognition/tensorflow_models/quantization/ptq/resnet50_v1.yaml (英語) ファイルを編集します。量子化と評価のデフォルトのデータセット・パス (root: /path/to/evaluation/dataset) を、前のデータの準備ステップで生成した TFRecord 形式のデータセットを保存する実際のローカルパスに変更します。

```
$ cd examples/tensorflow/image_recognition/tensorflow_models/quantization/ptq
$ bash run_tuning.sh --config=resnet50_v1.yaml \
    --input_model=/PATH/TO/resnet50_fp32_pretrained_model.pb \
    --output_model=./nc_resnet50_v1.pb
```

参考資料: https://github.com/intel/neural-compressor/tree/master/examples/tensorflow/image_recognition/tensorflow_models/quantization/ptq#1-resnet50-v10 (英語)

混合精度 (BF16 + FP32) の実行

「混合精度」(英語) を参照してください。

精度のチューニングなしで変換

この方法では、できるだけ多くのノードを BF16 に変換します。そのため、モデルの精度が低下することが予想されます。

```
$ python convert_bf16_without_tuning.py

from neural_compressor.experimental import MixedPrecision
converter = MixedPrecision()
converter.precisions = 'bf16'
converter.model = '/PATH/TO/resnet50_fp32_pretrained_model.pb'
optimized_model = converter()
optimized_model.save('nc_resnet50_v1_bf16.pb')
```

精度のチューニングありで変換

この方法では精度を重視するため、事前に評価が必要です。予想どおりモデルの精度は低下しますが、上記の方法と比較して BF16 に変換されるノードが少なくなります。

1. `resnet50_v1.yaml` を編集します。

   以下のコードを追加します。

   ```
   mixed_precision:
     precisions: 'bf16'
   ```

   次のようになります。

   ```
   model:
     name: resnet50_v1
     framework: tensorflow

   mixed_precision:
     precisions: 'bf16'

   evaluation:
     accuracy:
       dataloader:
         ...
       metric:
         ...
   ```

2. Python* を実行します。

   ```
   $ python convert_bf16_with_tuning.py

   from neural_compressor.experimental import MixedPrecision
   converter = MixedPrecision('resnet50_v1.yaml')
   converter.precisions = 'bf16'
   converter.model = '/PATH/TO/resnet50_fp32_pretrained_model.pb'
   optimized_model = converter()
   optimized_model.save('nc_resnet50_v1_bf16.pb')
   ```

INT8 + BF16 + FP32

モデル内で INT8、BF16、FP32 を混在させ、パフォーマンスを最適化することが可能です。この例では、量子化モデルの `nc_resnet50_v1.pb` に混合精度を適用し、INT8 + BF16 + FP32 のモデルにします。

```
$ python convert_bf16_without_tuning.py

from neural_compressor.experimental import MixedPrecision
converter = MixedPrecision()
converter.precisions = 'bf16'
converter.model = '/PATH/TO/nc_resnet50_v1.pb'
optimized_model = converter()
optimized_model.save('nc_resnet50_v1_int8_bf16.pb')
```

ログを確認すると、MatMul という 1 つの演算を BF16 に変換できることが分かります。これにより、パフォーマンスが少し向上するでしょう。量子化モデルにほかの FP32 演算がある場合は、混合精度で BF16 に変換して、パフォーマンスを向上できる可能性があります。

```
2022-08-04 19:50:01 [INFO] |**********Mixed Precision Statistics**********|
2022-08-04 19:50:01 [INFO] +-------------+--------+-------+-------+-------+
2022-08-04 19:50:01 [INFO] |   Op Type   | Total  |  INT8 |  BF16 |  FP32 |
2022-08-04 19:50:01 [INFO] +-------------+--------+-------+-------+-------+
2022-08-04 19:50:01 [INFO] |   MaxPool   |   1    |   1   |   0   |   0   |
2022-08-04 19:50:01 [INFO] |   AvgPool   |   1    |   1   |   0   |   0   |
2022-08-04 19:50:01 [INFO] |    Conv2D   |   53   |   53  |   0   |   0   |
2022-08-04 19:50:01 [INFO] |    MatMul   |   1    |   0   |   1   |   0   |
2022-08-04 19:50:01 [INFO] |  QuantizeV2 |   1    |   1   |   0   |   0   |
2022-08-04 19:50:01 [INFO] |  Dequantize |   1    |   1   |   0   |   0   |
2022-08-04 19:50:01 [INFO] |     Cast    |   2    |   0   |   1   |   1   |
2022-08-04 19:50:01 [INFO] +-------------+--------+-------+-------+-------+
```

ベンチマークの実行

パフォーマンス・モードでベンチマークを実行します。

```
$ bash run_benchmark.sh --input_model=./xxx.pb --config=resnet50_v1.yaml --
mode=performance
```

パフォーマンス・モード のベンチマーク結果は次のようになります。

```
Batch size = 1
Latency: xxx
Throughput: xxx
```

精度モードでベンチマークを実行します。

```
$ bash run_benchmark.sh --input_model=./xxx.pb --config=resnet50_v1.yaml --
mode=accuracy
```

精度モード のベンチマーク結果は次のようになります。

```
Accuracy is x.xx
Batch size = 32
```

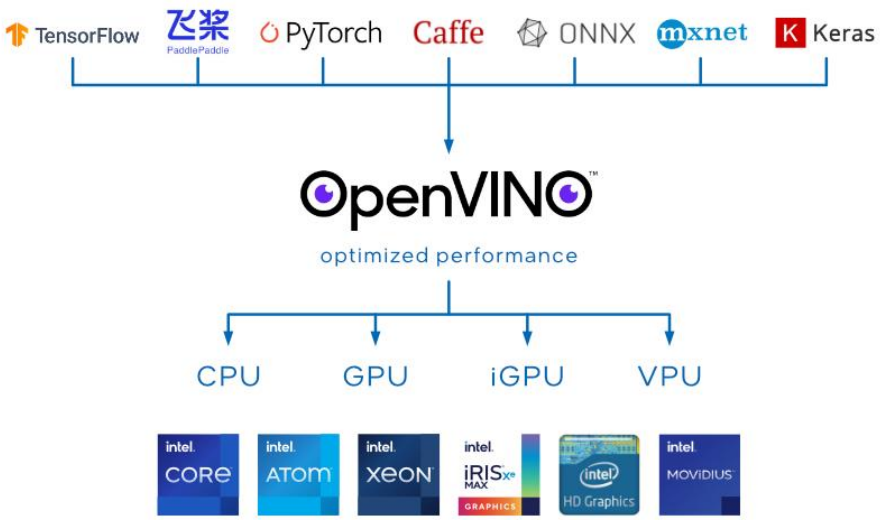
インテル® ディストリビューションの OpenVINO™ ツールキットによる推論の高速化

インテル® ディストリビューションの OpenVINO™ ツールキット

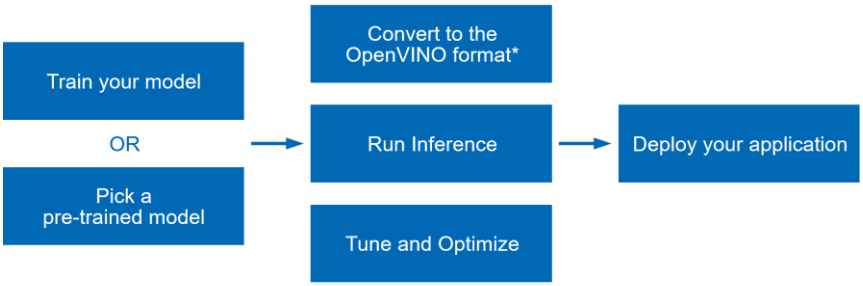
OpenVINO™ ツールキットはオープンソースであり、以下の特徴を持つ AI 推論を最適化して展開します。

- コンピューター・ビジョン、自動音声認識、自然言語処理、その他の一般的なタスクにおけるディープラーニングのパフォーマンスを向上します。
- TensorFlow*、PyTorch* などの一般的なフレームワークでトレーニングしたモデルを使用できます。

- エッジからクラウドまで、さまざまなインテル® プラットフォームでリソース要件を軽減し、効率良く展開できます。



以下の図は、OpenVINO™ ツールキットでトレーニングしたディープラーニング・モデルをデプロイする典型的なワークフローを示しています。



- TensorFlow* や PyTorch* などの一般的なフレームワークでモデルをトレーニングするか、Open Model Zoo からトレーニング済みモデルを取得します。
- モデル・オプティマイザーを実行してスタティック・モデル分析を行い、OpenVINO™ ランタイムで推論できる最適化されたモデルの中間表現 (IR) を作成します。
- OpenVINO™ ランタイム API を使用して、中間表現 (IR)、ONNX*、または PaddlePaddle モデルを読み込み、任意のデバイスで実行します。
- 量子化、プルーニング、前処理の最適化などを適用して、パイプライン全体のチューニングと最適化を行い、最終的なモデルのパフォーマンスを向上します。
- すべてが完了したら、OpenVINO™ ツールキットを使用してアプリケーションをデプロイ (英語) します。

詳細は、OpenVINO™ ツールキットのオンライン・ドキュメントを参照してください。
https://docs.openvino.ai/latest/index.html (英語)

インテル® DL ブーストで BF16/INT8 推論を実現

CPU プリミティブのデフォルトの浮動小数点精度は FP32 です。AVX512_BF16 または AMX_BF16 拡張で BF16 計算をネイティブにサポートするプラットフォームでは、FP32 の代わりに BF16 が自動的に使用され、パフォーマンスが向上します。BF16 の詳細は、「BFLOAT16 – ハードウェアの数値定義」(英語) を参照してください。

BF16 精度を使用すると、以下のようなパフォーマンスの利点が得られます。

- BF16 データは仮数が短いため、2 つの BF16 数値の乗算はより高速です。
- BF16 のデータサイズは 32 ビット float の 1/2 であるため、メモリー使用量を軽減できます。

CPU デバイスが BF16 データ型をサポートしているか確認するには、openvino.runtime.Core.get_property (英語) を使用して、ov::device::capabilities (英語) プロパティーを照会します。以下のように、CPU 機能のリストに BF16 が含まれているはずです。

```
core = Core()
cpu_optimization_capabilities = core.get_property("CPU",
"OPTIMIZATION_CAPABILITIES")
```

benchmark_app を使用して、推論実行時に BF16/INT8 が有効かどうかを確認します。

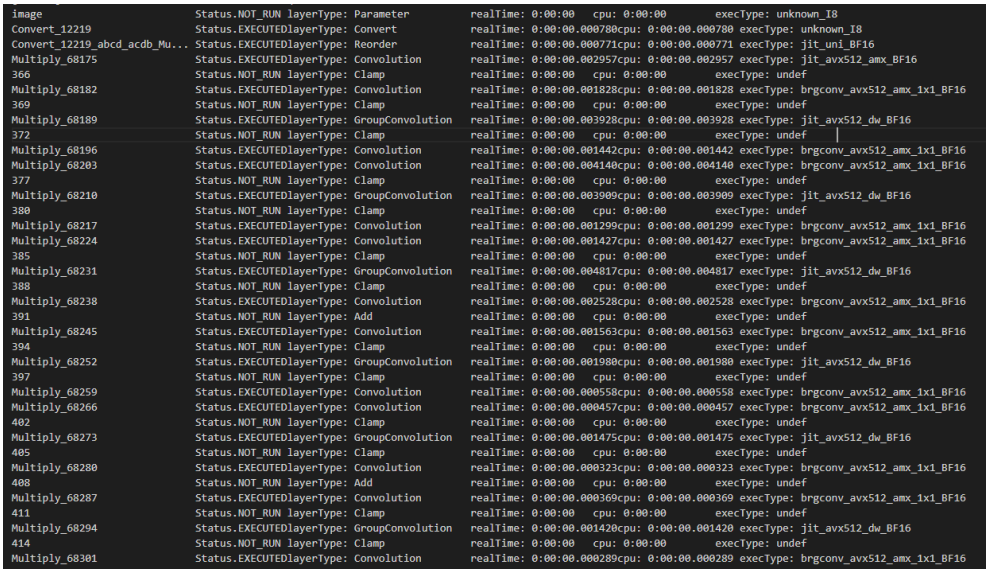
- OpenVINO™ 開発ツールをインストール (英語) して、OpenVINO™ ランタイムをインストール (英語) します。
- BF16
    - Open Model Zoo から FP32 モデルをダウンロードします (または独自の FP32 モデルを選択します)。ここでは例として、horizontal-text-detection-0001 (英語) をダウンロードします。

      ```
      $ omz_downloader --name horizontal-text-detection-0001 --precisions
      FP32 -o .
      ```

    - ベンチマーク・アプリケーションを -pc で実行します。

      ```
      $ benchmark_app -m ./intel/horizontal-text-detection-
      0001/FP32/horizontal-text-detection-0001.xml -pc
      ```

    - いくつかのカーネルが、インテル® AVX-512 命令とインテル® AMX 命令の両方で BF16 で実行していることが確認できます。



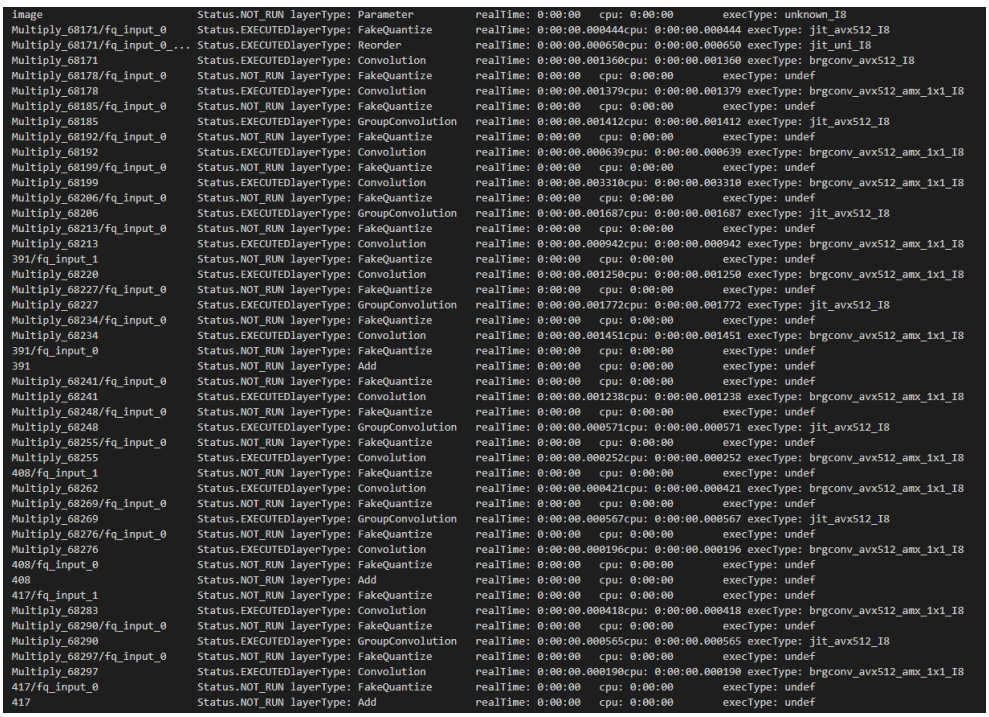
- INT8
  - Open Model Zoo から INT8 モデルをダウンロードします (または独自の INT8 モデルを選択します)。ここでは例として、horizontal-text-detection-0001 (英語) をダウンロードします。

    ```
    $ omz_downloader --name horizontal-text-detection-0001 --precisions FP16-INT8 -o .
    ```

  - ベンチマーク・アプリケーションを -pc で実行します。

    ```
    $ benchmark_app -m ./intel/horizontal-text-detection-0001/FP16-INT8/horizontal-text-detection-0001.xml -pc
    ```

  - いくつかのカーネルが、インテル® AVX-512 命令とインテル® AMX 命令の両方で INT8 で実行していることが確認できます。



注

BF16 データ型の仮数部はサイズが小さいため、特に BF16 でトレーニングされていないモデルでは、結果の推論精度が FP32 推論と異なる場合があります。BF16 の推論精度が許容できない場合は、FP32 精度に切り替えてください。

C++:
```
ov::Core core;
core.set_property("CPU", ov::hint::inference_precision(ov::element::f32));
```

Python*:
```
core = Core()
core.set_property("CPU", {"INFERENCE_PRECISION_HINT": "f32"})
```

例えば、`-infer_precision` を `f32` に設定して benchmark_app を使用します。

```
$ benchmark_app -m ./intel/horizontal-text-detection-0001/FP32/horizontal-text-detection-0001.xml -pc -infer_precision f32
```

データ・アナリティクスとマシンラーニングの高速化

人工知能 (AI) の一分野として、現在、マシンラーニングが注目されています。マシンラーニングを利用したアナリティクスも人気を集めています。ほかのアナリティクスと比較すると、マシンラーニングは IT スタッフ、データ・サイエンティスト、そしてさまざまな組織のビジネスチームが、AI の強みを素早く発揮するのに役立ちます。さらに、マシンラーニングは多くの新しい商用およびオープンソースのソリューションを提供しており、開発者に大きなエコシステムを提供しています。開発者は、scikit-learn*、Cloudera*、Spark* MLlib など、さまざまなオープンソースのマシンラーニング・ライブラリーから選択することができます。

インテル® ディストリビューションの Python*

インテル® ディストリビューションの Python* は、AI ソフトウェア開発者向けの Python* 開発ツールキットです。インテル® Xeon® スケーラブル・プロセッサー・プラットフォーム上で Python* の計算を高速化できます。Anaconda* で利用できるほか、Conda、**PIP**、APT GET、YUM、Docker* などでインストールして利用することが可能です。

インテル® ディストリビューションの Python* の機能:

- インテル® アーキテクチャー向けに最適化された命令セットを備えた、人気があり、急成長しているプログラミング言語の利点を活用できます。
- インテル® パフォーマンス・ライブラリーを使用して構築された Python* のコア数値演算および科学パッケージの高速化により、ネイティブに近いパフォーマンスを実現します。
- 効率良いマルチスレッド、ベクトル化、メモリー管理を実現し、科学計算をクラスター全体で効率的にスケールアップします。
- コアパッケージには、Numba、NumPy*、SciPy* などがあります。

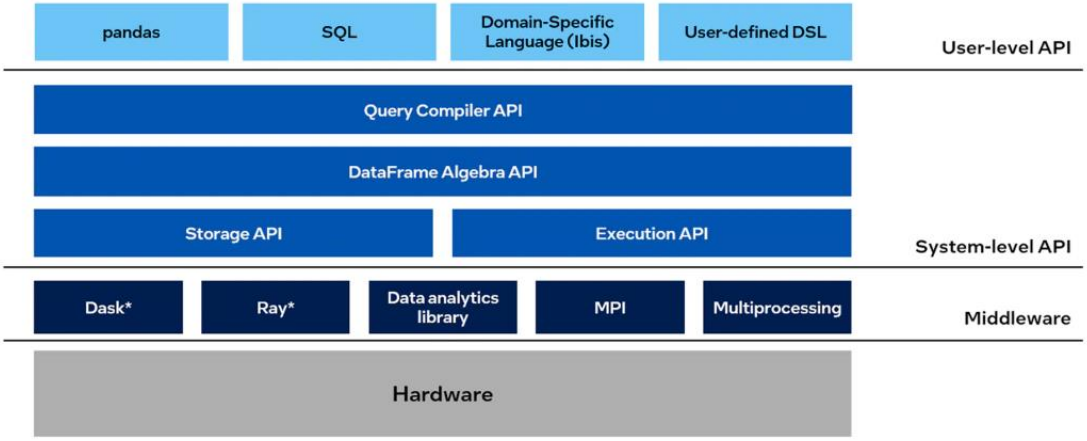
参考資料とダウンロード: https://www.intel.com/content/www/us/en/developer/tools/oneapi/distribution-for-python.html (英語)

Modin 向けインテル® ディストリビューション

Modin は pandas を置き換えるもので、データ・サイエンティストは API コードを変更することなく、分散 DataFrame 処理に拡張できます。Modin 向けインテル® ディストリビューションには、インテルのハードウェアでの処理をさらに高速化する最適化が追加されています。

このライブラリーを使用すると、以下のことが可能になります。

- 1 台のワークステーションでテラバイト単位のデータを処理
- 同じコードを使用して、1 台のワークステーションからクラウドまでスケールアップ
- 新しい API の習得ではなく、データ分析に注力



Modin 向けインテル® ディストリビューションの機能

- DataFrame 処理の高速化
  - 大規模な DataFrame の抽出、変換、ロード (ETL) プロセスを高速化します。
  - マシンで利用可能なすべてのプロセシング・コアを自動的に使用します。
- インテルのハードウェア向けの最適化
  - インテル® Optane™ パーシステント・メモリーを使用して、1 台のデータ・サイエンス・ワークステーション (英語) でテラバイト・データに拡張します。
  - HEAVY.AI* アナリティクスによる大規模データセット (10 億行以上) を分析します。
- 既存の API やエンジンとの互換性
  - 規模に関係なく、コードの 1 行を変更するだけで既存の pandas API 呼び出しを使用できます。pandas を pd としてインポートする代わりに、次のコマンドを使用して `modin.pandas` を `pd` としてインポートするだけです。

    ```
    import modin.pandas as pd

    # import pandas as pd
    import modin.pandas as pd
    df = pd.read_csv("my_dataset.csv")
    ```

  - Dask*、Ray、HEAVY.AI 計算エンジンを使用して、コードを記述することなくデータを分散できます。

- NumPy*、XGBoost、scikit-learn* など、Python* エコシステムの残りのコードは引き続き使用できます。
- 同じノートブックを使用して、ローカルマシンからクラウドへスケーリングします。

参考資料とダウンロード: https://www.intel.com/content/www/us/en/developer/tools/oneapi/distribution-of-modin.htm (英語)

scikit-learn* 向けインテル® エクステンション

scikit-learn* 向けインテル® エクステンションは、シングルノードおよびマルチノード構成のインテルの CPU および GPU 向けの scikit-learn* アプリケーションをシームレスに高速化します。この拡張パッケージは、マシンラーニング・アルゴリズムのパフォーマンスを向上しつつ、scikit-learn* の推定器に動的にパッチを適用します。

主なメリットは以下のとおりです。

- 新しい API を学習が不要
- Python* エコシステムとの統合
- 標準の scikit-learn と比較して最大 100 倍のパフォーマンスと精度を実現

scikit-learn* にパッチを適用する方法 (英語)

```
import numpy as np

# Turn on scikit-learn optimizations with these 2 simple lines:

from sklearnex import patch_sklearn

patch_sklearn()


# Import scikit-learn algorithms after the patch is enabled

from sklearn.cluster import KMeans


X = np.array([[1, 2], [1, 4], [1, 0],

 [10, 2], [10, 4], [10, 0]])

kmeans = KMeans(n_clusters=2, random_state=0).fit(X)

print(f"kmeans.labels_ = {kmeans.labels_}")
```

参考資料とダウンロード: https://www.intel.com/content/www/us/en/developer/tools/oneapi/scikit-learn.html (英語)

インテル® アーキテクチャー向けに最適化された XGBoost

XGBoost 0.81 以降でインテルは、インテル® CPU で優れたパフォーマンスを提供する最適化を直接追加してきました。この有名な勾配ブースティング決定木のマシンラーニング・パッケージに、インテル® アーキテクチャー向けのドロップイン・アクセラレーションが追加され、モデルのトレーニングを大幅にスピードアップし、より正確に予測するため精度を向上します。

参考資料とダウンロード: https://www.intel.com/content/www/us/en/developer/articles/technical/xgboost-optimized-architecture-getting-started.html (英語)

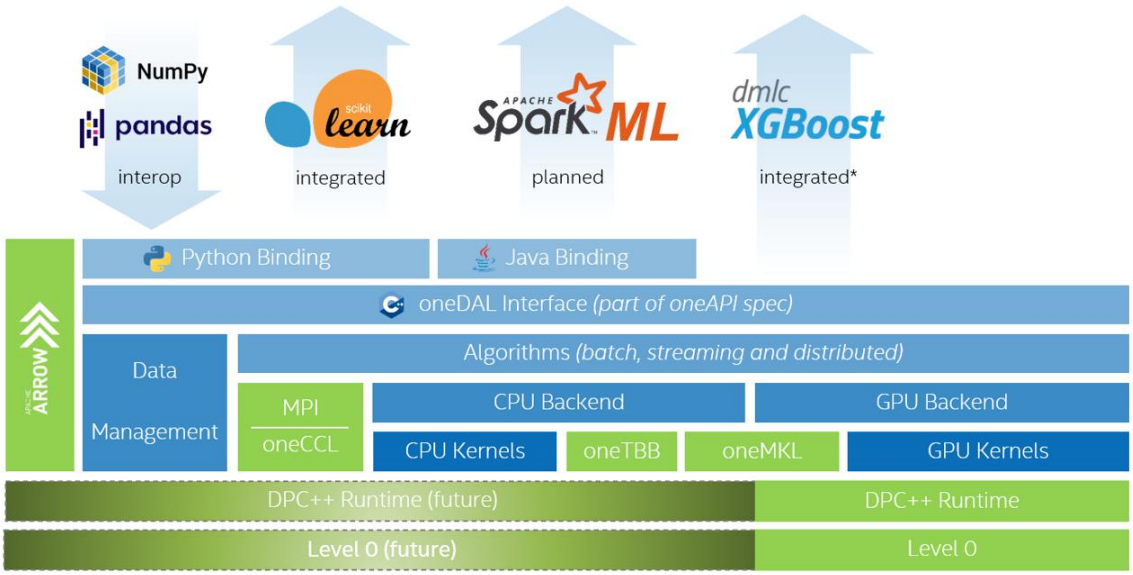
インテル® oneAPI データ・アナリティクス・ライブラリー (インテル® oneDAL)

インテル® oneDAL は、バッチ、オンライン、分散処理モードでのデータ解析 (前処理、変換、分析、モデリング、検証、意思決定) のすべての段階に高度に最適化されたアルゴリズム・ビルディング・ブロックを提供することで、ビッグデータ解析のスピードアップを支援するライブラリーです。

このライブラリーは、アルゴリズム計算とともにデータ取り込みを最適化し、スループットとスケーラビリティーを向上します。C++ および Java* の API と、Spark* や Hadoop* などの一般的なデータソースへのコネクターが含まれています。インテル® oneDAL の Python* ラッパーは、Python* 向けインテル® ディストリビューションの一部です。

典型的な機能に加えて、インテル® oneDAL は従来の C++ インターフェイスに DPC++ SYCL* API 拡張を提供し、一部のアルゴリズムで GPU を使用できるようにします。

このライブラリーは、特に分散計算に役立ちます。通信層から独立した分散アルゴリズム向けにビルディングブロックの完全なセットを提供します。これにより、ユーザーは選択した通信手段を使用して、高速でスケーラブルな分散アプリケーションを構築できます。



参考資料とダウンロード:
https://www.intel.com/content/www/us/en/developer/tools/oneapi/onedal.html (英語)

関連情報

インテル® AVX-512 に関する情報: https://colfaxresearch.com/skl-avx512/ (英語)
インテル® Optimized AI Frameworks:
https://www.intel.com/content/www/us/en/developer/tools/frameworks/overview.html (英語)

製品および性能に関する情報

[1] 性能は、使用状況、構成、その他の要因によって異なります。詳細については、http://www.intel.com/PerformanceIndex/ (英語) を参照してください。