

インテル® AMX を使用した PyTorch のトレーニングと推論の高速化

この記事は、インテルのウェブサイトで公開されている「[Accelerate PyTorch Training and Inference using Intel® AMX](#)」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。

PyTorch は、Torch ライブラリーをベースとするディープラーニング・フレームワーク (英語) であり、主に [コンピュータービジョン](#) (英語) や自然言語処理アプリケーションで使用されます。このフレームワークは Meta 社によって開発され、現在は [PyTorch Foundation](#) (英語) がサポートしています。インテルは、オープンソースの PyTorch プロジェクトと連携して、インテル® アーキテクチャー向けにフレームワークを最適化しています。最新の [最適化](#) (英語) と機能は、PyTorch の stock ディストリビューションにアップストリームされる前に、PyTorch 向けインテル® エクステンションでリリースされています。

この記事では、第 4 世代インテル® Xeon® プロセッサの AI アクセラレーター・エンジンである、[インテル® アドバンスド・マトリクス・エクステンション \(インテル® AMX\)](#) を紹介し、PyTorch を使用して AI トレーニングと [推論](#) (英語) のパフォーマンスを高速化する方法を説明します。

第 4 世代インテル® Xeon® スケーラブル・プロセッサのインテル® AMX

第 4 世代インテル® Xeon® プロセッサは、パフォーマンスの向上、電力効率の良いコンピューティング、セキュリティの強化を実現するとともに、総所有コストを最適化するように設計されているため、CPU の AI ワークロードへの適用範囲が広がります。

インテル® AMX は、第 4 世代インテル® Xeon® プロセッサの各コアに内蔵されているアクセラレーターで、ディープラーニングのトレーニングと推論のワークロードを高速化します。インテル® AMX アーキテクチャーは、2 つの主要コンポーネントで構成されています。

1. タイル - サイズが 1KB の、新しい拡張可能な 2D レジスターファイル。
2. TMUL (Tile Matrix Multiply、タイル行列乗算) - タイルを操作して AI 向けの行列乗算計算を行う命令。



簡単に言えば、インテル® AMX は各コアに大量のデータを格納して、1 回の操作で大規模な行列を計算します。インテル® AMX は BF16 と INT8 データ型のみをサポートしています。FP32 データ型は、第 3 世代インテル® Xeon® プロセッサに搭載されているインテル® アドバンスド・ベクトル・エクステンション 512 (インテル® AVX-512) 命令によって引き続きサポートされます。インテル® AMX は、レコメンダー・システム、自然言語処理、画像検出などのディープラーニング・ワークロードを高速化します。表形式データを使用する**古典的なマシンラーニング・ワークロード** (英語) では、既存のインテル® AVX-512 命令が使用されます。**多くのディープラーニング・ワークロードは混合精度であり、第 4 世代インテル® Xeon® プロセッサはインテル® AMX とインテル® AVX-512 をシームレスに移行して、最も効率的な命令セットを使用してコードを実行できます。**

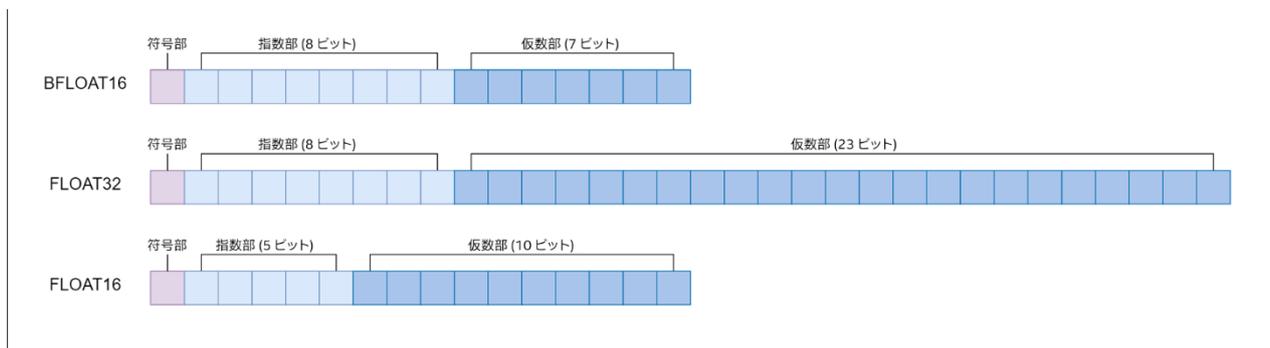
混合精度学習とは?

混合精度学習とは、メモリー使用量を減らして高速に実行できるよう、モデルのパラメーターを異なる精度のデータ型 (FP16 と FP32 が最も一般的です) に保存して、大規模なニューラル・ネットワークをトレーニングする手法です。

現在のほとんどのモデルは、32 ビットのメモリーを必要とする単精度浮動小数点 (FP32) データ型を使用しています。しかし、精度の低い float16 と bfloat16 (BF16) の 2 つのデータ型では、メモリー使用量はそれぞれ 16 ビットで済みます。bfloat16 は 16 ビットのコンピューター・メモリーを占有する浮動小数点形式で、32 ビット浮動小数点数のダイナミック・レンジをほぼ表現できます。bfloat16 形式は次のように表現されます。

- 1 ビット - 符号部
- 8 ビット - 指数部
- 7 ビット - 仮数部

float16、float32、bfloat16 の比較を図に示します。



PyTorch 向けインテル® エクステンション

このインテルの拡張機能は、最新の機能と最適化によって PyTorch を拡張し、インテルのハードウェアのパフォーマンスをさらに向上します。これらの新機能のほとんどは、PyTorch の stock 実装の将来のバージョンにアップストリームされます。拡張機能は、スタンドアロンのコンポーネント (英語) として、または [インテル® AI アナリティクス・ツールキット \(英語\)](#) の一部として提供されます。(インテルの拡張機能のインストール方法は、[インストール・ガイド \(英語\)](#) を参照してください。)

拡張機能は、Python モジュールとしてロードすることも、C++ ライブラリーとしてリンクすることもできます。Python ユーザーは、`intel_extension_for_pytorch` をインポートすることにより、動的に有効にできます。

- [CPU チュートリアル \(英語\)](#) では、インテル® CPU 向けの PyTorch 向けインテル® エクステンションに関する詳細情報を提供しています。ソースコードは、[main ブランチ \(英語\)](#) から入手できます。
- [GPU チュートリアル \(英語\)](#) では、インテル® GPU 向けの PyTorch 向けインテル® エクステンションに関する詳細情報を提供しています。ソースコードは、[xpu-main ブランチ \(英語\)](#) から入手できます。

PyTorch モデルでインテル® AMX の bfloat16 混合精度学習を有効にする方法

最初のステップは、ハードウェアでインテル® AMX が有効であるかを確認することです。bash ターミナルで、次のコマンドを入力します。

```
cat /proc/cpuinfo
```

次のコマンドも使用できます。

```
lscpu | grep amx
```

「flah」セクションで `amx_bf16` と `amx_int8` を確認します。見つからない場合は、Linux カーネル 5.17 以降にアップグレードしてください。出力は次のようになります。

```
Flags:            fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse s
se2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpu
id aperfmperf tsc_known_freq pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 s
se4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb cat_l3 cat_
l2 cdp_l3 invpcid_single intel_ppin cdp_l2 ssbd mba ibrs ibpb stibp ibrs_Enhanced tpr_shadow vnmi flexpriority ept vpid ept_
ad fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm cqm rdt_a avx512f avx512dq rdseed adx smap avx512ifma clflus
hopt clwb intel_pt avx512cd sha_ni avx512bw avx512vl xsaveopt xsavec xgetbv1 xsaves cqm_llc cqm_occup_llc cqm_mbm_total cqm_
mbm_local split_lock_detect avx_vnni avx512_bf16 wbnoinvd dtherm ida arat pln pts hwp hwp_act_window hwp_epp hwp_pkg_req hfi
avx512vbmi umip pku ospke waitpkg avx512_vbmi2 gfni vaes vpclmulqdq avx512_vnni avx512_bitalg tme avx512_vpocntdq la57 rdp
id bus_lock_detect cldemote movdiri movdir64b enqcmd fsrm uintr avx512_vp2intersect md_clear serialize tsxldtrk pconfig arch
lbr amx_bf16 avx512_fp16 amx_tile amx_int8 flush_lid arch_capabilities
```

PyTorch 向け Intel® エクステンションは、実行時に機能を自動的に検出してコードをディスパッチします。これは、オープンソースのクロスプラットフォーム・ライブラリーである Intel® oneAPI ディープ・ニューラル・ネットワーク・ライブラリー (Intel® oneDNN (英語)) と似ています。このライブラリーは、ディープラーニング・ビルディング・ブロックの最適化された実装を提供し、すでに使用しているフレームワークのパフォーマンスを向上します。環境変数 `ONEDNN_MAX_CPU_ISA` を実行時に設定して命令セット・アーキテクチャー (ISA) を変更できますが、デフォルトは最新の Intel® AMX であるため、設定する必要はありません。すべてのオプションのリストは、oneDNN ドキュメントの [CPU ディスパッチャー・コントロール \(英語\)](#) を参照してください。

PyTorch で Intel® AMX の BF16 を有効にする手順

数行のコードで有効にできます。PyTorch 向け Intel® エクステンション・ライブラリーをインポートして、`torch.bfloat16` データ型を `optimize()` 関数に渡します。最後に、`torch.cpu.amp.autocast()` 関数を使用します。ResNet50 でのトレーニングと推論については、次の例を参照してください。

トレーニングのサンプルコード

```
import torch
import torchvision
# ライブラリーをインポートします。torch または torchvision のインポート直後が最適です。
import intel_extension_for_pytorch as ipex

LR = 0.001
DOWNLOAD = True
DATA = '&#x27;datasets/cifar10/&#x27;

transform = torchvision.transforms.Compose([
    torchvision.transforms.Resize((224, 224)),
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
train_dataset = torchvision.datasets.CIFAR10(
    root=DATA,
    train=True,
    transform=transform,
    download=DOWNLOAD,
)
train_loader = torch.utils.data.DataLoader(
    dataset=train_dataset,
    batch_size=128
)

model = torchvision.models.resnet50()
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr = LR, momentum=0.9)
model.train()

# ipex.optimize() はモデルを最適化するためのメイン関数です。データ型も指定できます。
optimizer = ipex.optimize(model, optimizer=optimizer, dtype=torch.bfloat16)

for batch_idx, (data, target) in enumerate(train_loader):
    optimizer.zero_grad()
```

PyTorch 向けインテル® エクステンションを使用するかどうかに関係なく、混合精度を使用するには、この行が必要です。

```
with torch.cpu.amp.autocast():
    output = model(data)
    loss = criterion(output, target)
    loss.backward()
    optimizer.step()
    print(batch_idx)
torch.save({
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
}, 'checkpoint.pth')
```

推論のサンプルコード

```
import torch
import torchvision.models as models

model = models.resnet50(weights='ResNet50_Weights.DEFAULT')
model.eval()
data = torch.rand(1, 3, 224, 224)
```

```
##### code changes #####
import intel_extension_for_pytorch as ipex
model = ipex.optimize(model, dtype=torch.bfloat16)
#####
```

混合精度の場合は `torch.cpu.amp.autocast()` をインクルードします。推論の場合は、パフォーマンス向上のため TorchScript のグラフモードでモデルを実行することを強く推奨します。

```
with torch.no_grad(), torch.cpu.amp.autocast():
    model = torch.jit.trace(model, torch.rand(1, 3, 224, 224))
    model = torch.jit.freeze(model)
```

```
model(data)
```

`torch.bfloat16` を `optimize()` 関数に渡すと、モデルのパラメーターが BF16 にキャストされます。`torch.cpu.amp.autocast()` は、混合精度 (このケースでは BF16) で演算を実行します。

インテル® AMX を使用したトレーニングと推論の最適化

トレーニング

この[サンプルコード](#) (英語) は、PyTorch 向けインテル® エクステンションで CIFAR10 データセットを使用して ResNet50 モデルをトレーニングする方法を示しています。FP32 よりもインテル® AMX BF16 でパフォーマンスが向上しています。

サンプルコードでは、次の手順を実装しています。

1. `cpuinfo` のフラグをチェックして、ハードウェアがインテル® AMX をサポートしているかどうかを確認します。

2. CIFAR10 データセットをロードします。
3. 環境変数 **ONEDNN_MAX_CPU_ISA** を設定します。
 - a. DEFAULT - インテル® AMX で実行している場合
 - b. AVX512_CORE_BF16 - インテル® AVX-512 で実行している場合
4. ResNet50 モデルをインスタンス化して、選択したモデルとトレーニング・オプティマイザーで PyTorch 向けインテル® エクステンションの `optimize()` 関数を使用します。
5. 該当する場合は混合精度を使用して、次の実行ケースでモデルをトレーニングします。トレーニング時間を記録します。
 - a. FP32 (ベースライン)
 - b. BF16 (インテル® AVX-512)
 - c. BF16 (インテル® AMX)
6. トレーニング時間を比較し、ベースラインの FP32 を基準として、すべての実行ケースのスピードアップを計算します。

第 4 世代インテル® Xeon® スケーラブル・プロセッサを使用して、[Linux 環境](#) (英語) または [インテル® Tiber™ AI クラウド](#) でサンプルコードを試してください。

推論

この[サンプルコード](#) (英語) は、PyTorch 向けインテル® エクステンションで ResNet50 および BERT モデルを使用して推論を実行する方法を示しています。FP32 よりもインテル® AMX BF16 および INT8 でパフォーマンスが向上しています。インテル® AMX INT8 と、INT8 操作の以前の命令セットであるインテル® AVX-512 ベクトル・ニューラル・ネットワーク命令 (VNNI) INT8 との比較もあります。

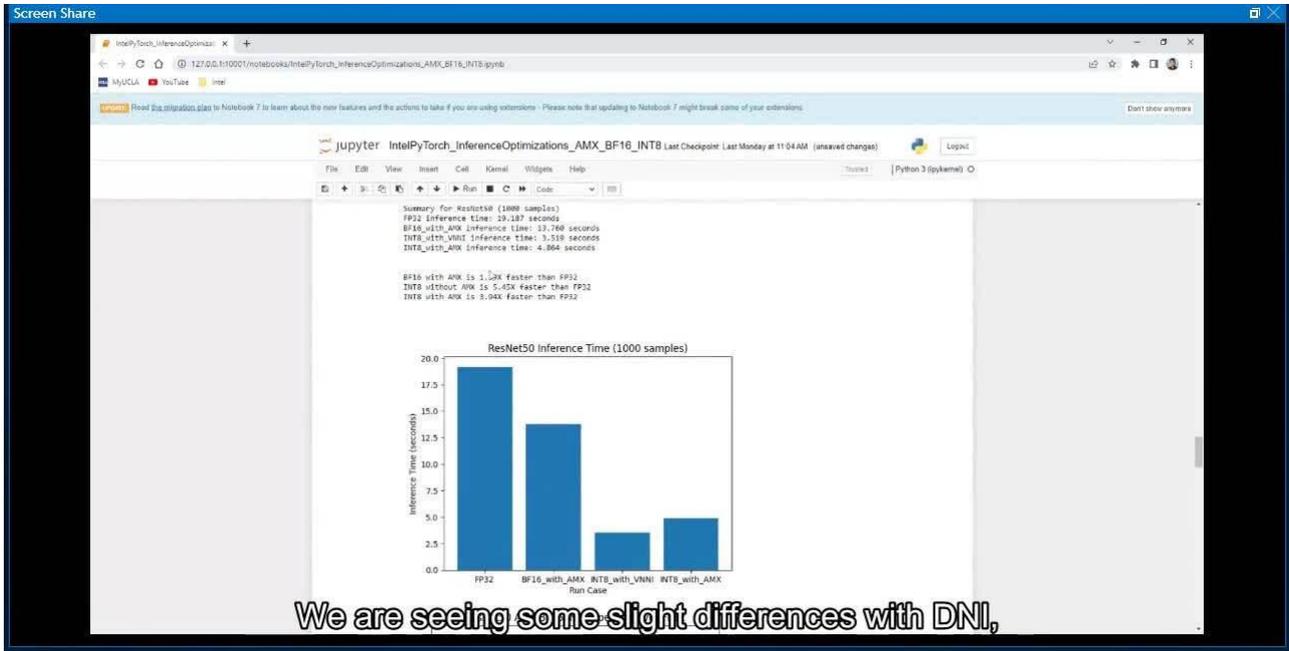
サンプルコードでは、次の手順を実装しています。

1. `cpuinfo` のフラグをチェックして、ハードウェアがインテル® AMX をサポートしているかどうかを確認します。
2. ResNet50 または BERT モデルをインスタンス化します。
3. 環境変数 **ONEDNN_MAX_CPU_ISA** を設定します。
 - a. DEFAULT - インテル® AMX で実行している場合
 - b. AVX512_CORE_VNNI - インテル® AVX-512 で実行している場合
4. 該当する場合は混合精度を使用して、次の実行ケースでモデルの推論を実行します。推論時間を記録します。INT8 を使用した実行ケースでは、オリジナルの FP32 モデルは PyTorch 向けインテル® エクステンションの量子化機能を使用して量子化しています。次に、グラフの最適化を活用するため、TorchScript を使用してすべてのモデルを JIT トレースしています。これは、モデルを本番環境にデプロイするときに役立ちます。
 - a. FP32 (ベースライン)
 - b. BF16 (インテル® AMX)
 - c. INT8 (インテル® AVX-512 VNNI INT8)
 - d. INT8 (インテル® AMX)
5. 推論時間を比較し、ベースラインの FP32 を基準として、すべての実行ケースのスピードアップを計算します。

第 4 世代インテル® Xeon® スケーラブル・プロセッサを使用して、[Linux 環境](#) (英語) または [インテル® Tiber™ AI クラウド](#) でサンプルコードを試してください。

関連するビデオを視聴する

最新のインテル® Xeon プロセッサでディープラーニングのワークロードを強化する方法の詳細は、次のビデオをご覧ください。



次のステップ

PyTorch 向けインテル® エクステンションを利用して、インテル® AMX を使用した第 4 世代インテル® Xeon® スケーラブル・プロセッサでの PyTorch のトレーニングと推論のパフォーマンスを高速化しましょう。

AI ソリューションの準備、構築、デプロイ、スケーリングを支援する、インテルのほかの [AI/ML フレームワークの最適化 \(英語\)](#) や [ツールのエンドツーエンドのポートフォリオ \(英語\)](#) をチェックして AI ワークフローに組み込み、インテルの [AI ソフトウェア・ポートフォリオ \(英語\)](#) の基盤である、統一されたオープンな標準ベースの [oneAPI プログラミング・モデル](#) について理解を深めることを推奨します。

インテルは、第 4 世代インテル® Xeon® スケーラブル・プロセッサ上でエンドツーエンドの AI パイプラインを実行できるように開発者を支援しています。詳細は、[インテルの AI ソリューション・プラットフォーム・ポータル \(英語\)](#) を参照してください。

関連資料

- [インテルの AI 開発ツールとリソース](#)
- [oneAPI 統一プログラミング・モデル](#)
- [公式サイト - インテルの PyTorch 最適化 \(英語\)](#)
- [PyTorch 向けインテル® エクステンション - ドキュメント \(英語\)](#)
- [インテル® アドバンスド・マトリクス・エクステンションの概要](#)
- [インテル® AMX による AI ワークロードの高速化](#)