インテル® Core™ Ultra プロセッサー上でインテル® VTune™ プロファイラーを使用して NPU 依存の AI PC アプリケーションを高速化

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 NPU-Bound AI PC Applications using Intel® VTune™ Profiler on Intel® Core™ Ultra Processor」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。

インテル® Core™ Ultra プロセッサーは、業界をリードするエンドユーザー体験を提供する 3 つの重要なコンピューティング・エンジンで構成されています。これら 3 つのコンピューティング・エンジンはそれぞれ、AI ワークロードの高速化の中核を担っています。ニューラル・プロセシング・ユニット (NPU) は、インテル® Core™ Ultra プロセッサーに搭載されている AI アクセラレーターであり、コンピューティング・アクセラレーションとデータ転送機能を備えた独自のアーキテクチャーを特徴としています。コンピューティング・アクセラレーションは、AI 演算用のハードウェア・アクセラレーション・ブロックで構成されるニューラル・コンピュート・エンジンによって実現されます。

NPU は、長時間にわたって持続的に実行される低消費電力の AI 支援ワークロードに最適です。例えば、背景のぼかしや画像のセグメント化などのウェブカメラ・タスクを実行する場合、NPU で実行すると全体のパフォーマンスが向上します。

AI アプリケーションのパフォーマンスを最大限に高めるには、NPU のコンピューティング・リソースとメモリー・リソースを最大限に活用する必要があります。この簡単なチュートリアルでは、その仕組みを説明します。

インテル® VTune™ プロファイラーを使用して、NPU で実行されているコードのボトルネックを特定できます。

- NPU と DDR メモリー間で転送されるデータ量の把握
- NPU で最も時間のかかるタスクの特定
- NPU の使用率を時間経過とともに視覚化
- NPU の帯域幅使用率の把握

このチュートリアルでは、インテル® VTune™ プロファイラーを NPU 解析用に設定し、結果を解釈する手順を詳しく説明します。

解析フロー

1. Python 環境を作成する
2. NPU 解析向けにインテル® VTune™ プロファイラーを設定する
3. 結果を解釈する
    - サマリービュー (NPU 全般解析)
    - ボトムアップ・ビュー
    - タイムライン・ビュー
4. 結果に基づいてワークロードを最適化する

サンプルコード

このチュートリアルでは、GitHub (英語) にある MobileNet v2 OpenVINO™ モデルのトレーニング後の量子化を使用します。セットアップと設定の詳細については、README (英語) を参照してください。

このサンプルは、Neural Network Compression Framework (NNCF) の Post-Training Quantization API を使用して、Imagenette (英語) データセットで事前トレーニング済みの MobileNet v2 (英語) 量子化の例に基づいて OpenVINO™ モデルを量子化します。

1. Python 環境を作成する

ワークロードに応じて Python 環境を作成します。

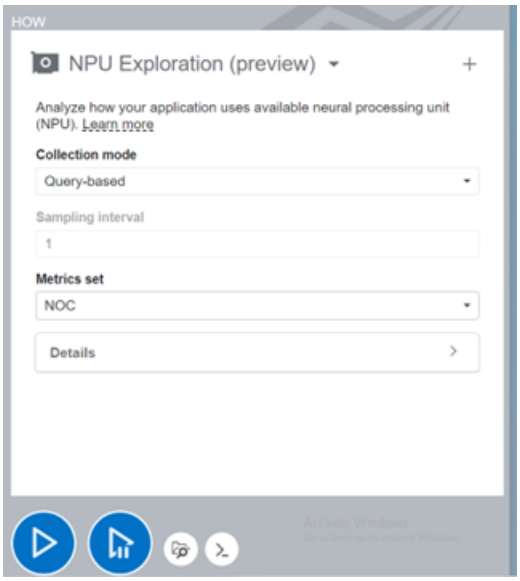
```
cd nncf/examples/post_training_quantization/openvino/mobilenet_v2/
pip install -r requirements.txt
```

2. NPU 解析向けにインテル® VTune™ プロファイラーを設定する

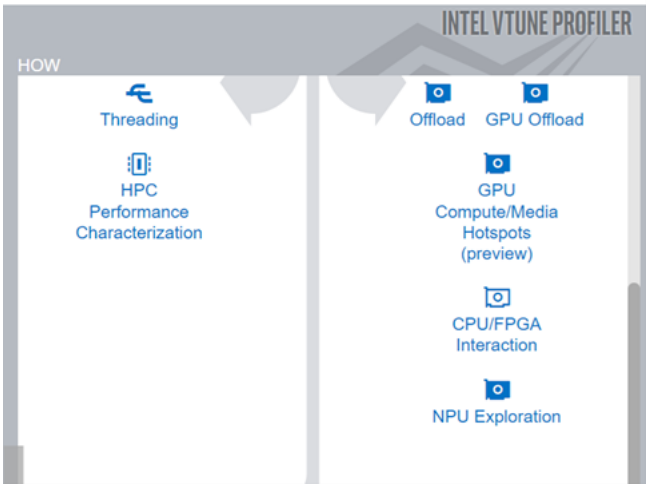
NPU 解析を行うには、**[Analysis Options (解析オプション)]** リストから **[NPU Exploration (NPU 全般解析)]** を選択します。

2 つのデータ収集モードがあり、それぞれ目的が異なります。

- Time-based (時間ベース): システム全体のメトリックを収集します。
  - 大規模なワークロード向け
  - オーバーヘッドが小さい
- Query-based (クエリーベース): レベルゼロ・インスタンスの各インスタンスのメトリックを収集します。
  - 小規模なワークロード向け
  - オーバーヘッドが大きい



現在サポートされている **[Metric Set (メトリックセット)]** は、NOC (ネットワークオンチップ) ファブリックのみです。NPU は、このファブリックを介して DDR メモリーとの間でデータの読み書きを行います。サンプリング間隔も、ワークロードと必要なプロファイル粒度に応じて設定できます。**[How (どのように)]** ペインからその他の詳細オプションを設定して、より細かく制御することも可能です。



この解析はコマンドラインから実行することもできます。以下に NPU 全般解析のコマンド例を示します。

一般的な形式:

```
vtune -collect npu [-knob <knob_name=knob_option>] -- <target> [target_options]
```

この例では、NPU 上で mobilenet-v2 モデルを実行し、インテル® VTune™ プロファイラーの NPU 全般解析を使用してアプリケーションをプロファイルします。

元のサンプルコード

mobilnet_unoptimized.py:

```
import re
import subprocess
from pathlib import Path
from typing import List

import numpy as np
import openvino as ov
import torch
from fastdownload import FastDownload
from rich.progress import track
from sklearn.metrics import accuracy_score
from torchvision import datasets
from torchvision import transforms

import nncf

ROOT = Path(__file__).parent.resolve()
DATASET_PATH = Path().home() / ".cache" / "nncf" / "datasets"
MODEL_PATH = Path().home() / ".cache" / "nncf" / "models"
MODEL_URL =
"https://huggingface.co/alexsu52/mobilenet_v2_imagenette/resolve/main/openvino_model.tgz"
DATASET_URL = "https://s3.amazonaws.com/fast-ai-imageclas/imagenette2-320.tgz"
DATASET_CLASSES = 10
```

```
def download(url: str, path: Path) -> Path:
    downloader = FastDownload(base=path.resolve(), archive="downloaded", data="extracted")
    return downloader.get(url)


def validate(model: ov.Model, val_loader: torch.utils.data.DataLoader) -> float:
    predictions = []
    references = []

    compiled_model = ov.compile_model(model, device_name="NPU")
    output = compiled_model.outputs[0]

    for images, target in track(val_loader, description="Validating"):
        pred = compiled_model(images)[output]
        predictions.append(np.argmax(pred, axis=1))
        references.append(target)

    predictions = np.concatenate(predictions, axis=0)
    references = np.concatenate(references, axis=0)
    return accuracy_score(predictions, references)


def run_benchmark(model_path: Path, shape: List[int]) -> float:
    cmd = ["benchmark_app", "-m", model_path.as_posix(), "-d", "GPU", "-api", "async", "-
niter", "50000", "-shape", str(shape)]
    cmd_output = subprocess.check_output(cmd, text=True)  # nosec
    print(*cmd_output.splitlines()[-8:], sep="\n")
    match = re.search(r"Throughput\: (.+?) FPS", cmd_output)
    return float(match.group(1))


def get_model_size(ir_path: Path, m_type: str = "Mb") -> float:
    xml_size = ir_path.stat().st_size
    bin_size = ir_path.with_suffix(".bin").stat().st_size
    for t in ["bytes", "Kb", "Mb"]:
        if m_type == t:
            break
        xml_size /= 1024
        bin_size /= 1024
    model_size = xml_size + bin_size
    print(f"Model graph (xml):   {xml_size:.3f} Mb")
    print(f"Model weights (bin): {bin_size:.3f} Mb")
    print(f"Model size:          {model_size:.3f} Mb")
    return model_size


# ##############################################################################
# Create an OpenVINO model and dataset

dataset_path = download(DATASET_URL, DATASET_PATH)

normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
val_dataset = datasets.ImageFolder(
    root=dataset_path / "val",
    transform=transforms.Compose(
        [
            transforms.Resize(256),
            transforms.CenterCrop(224),
            transforms.ToTensor(),
            normalize,
        ]
    ),
)
val_data_loader = torch.utils.data.DataLoader(val_dataset, batch_size=1, shuffle=False)

path_to_model = download(MODEL_URL, MODEL_PATH)
ov_model = ov.Core().read_model(path_to_model / "mobilenet_v2_fp32.xml")
```

```
###############################################################################
# Quantize an OpenVINO model
#
# The transformation function transforms a data item into model input data.
#
# To validate the transform function use the following code:
# >> for data_item in val_loader:
# >>    model(transform_fn(data_item))


def transform_fn(data_item):
    images, _ = data_item
    return images

calibration_dataset = nncf.Dataset(val_data_loader, transform_fn)
ov_quantized_model = nncf.quantize(ov_model, calibration_dataset)

###############################################################################
# Benchmark performance, calculate compression rate and validate accuracy

fp32_ir_path = ROOT / "mobilenet_v2_fp32.xml"
ov.save_model(ov_model, fp32_ir_path, compress_to_fp16=False)
print(f"[1/7] Save FP32 model: {fp32_ir_path}")
fp32_model_size = get_model_size(fp32_ir_path)

print("[3/7] Benchmark FP32 model:")
fp32_fps = run_benchmark(fp32_ir_path, shape=[1, 3, 224, 224])
```

コマンドラインの例:

```
vtune -c npu --  python mobilnet_unoptimized.py
```

3. 結果を解釈する

インテル® VTune™ プロファイラー GUI を起動し、`.vtune` ファイルを開くと、NPU 全般解析の結果を確認できます。

Top Task Overview (上位タスクの概要)

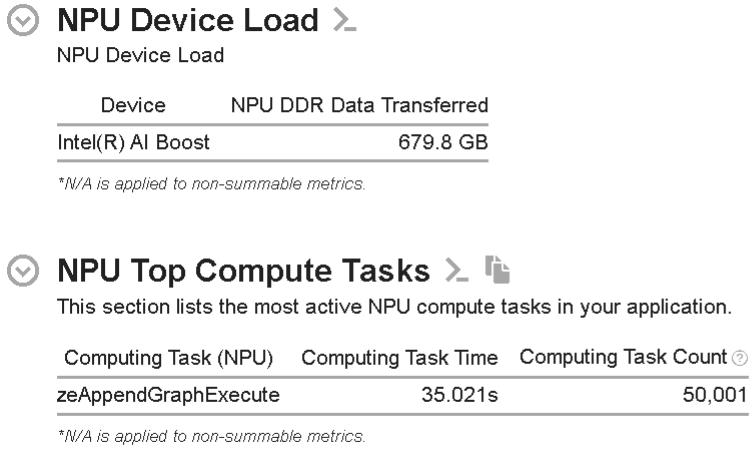
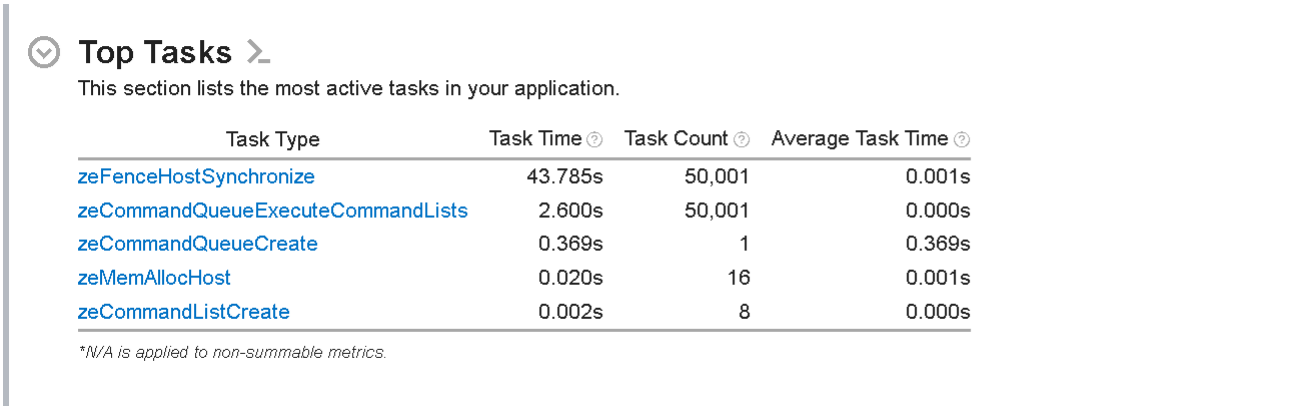


図 1: NPU 全般解析の **[Summary (サマリー)]** ビュー

[NPU Device Load (NPU デバイスロード)] セクションには、NOC を介して DDR SDRAM から NPU に転送されたデータが表示されます。図 1 に示すように、転送されたデータ量は 679.80GB です。

また、**[NPU Top Compute Tasks (NPU の上位計算タスク)]** セクションには、NPU で実行されている上位の計算タスクがリストされています。この例では、`zeAppendGraphExecute` によってグラフ実行コマンドがコマンドリストに追加されています。この API 呼び出しは、50,001 個の計算タスクを約 35 秒で処理しています。ここでグラフは、NPU で実行できる一連の操作を表します。



Top Tasks

This section lists the most active tasks in your application.

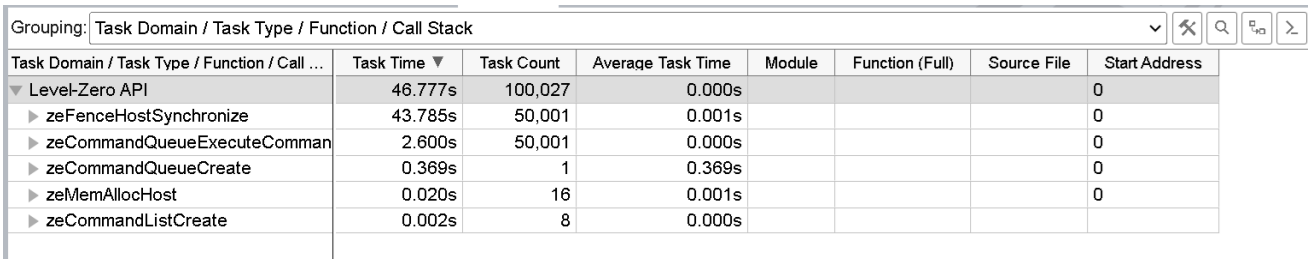
Task Type	Task Time	Task Count	Average Task Time
zeFenceHostSynchronize	43.785s	50,001	0.001s
zeCommandQueueExecuteCommandLists	2.600s	50,001	0.000s
zeCommandQueueCreate	0.369s	1	0.369s
zeMemAllocHost	0.020s	16	0.001s
zeCommandListCreate	0.002s	8	0.000s

*N/A is applied to non-summable metrics.

図 2: ホスト側の **[Top Tasks (上位タスク)]** リスト

[Top Tasks (上位タスク)] セクションには、アプリケーションのホスト側で実行されている最もアクティブなタスクがリストされています。このセクションは、CPU 時間を最も消費している関数またはコード領域を特定するのに役立ちます。最も時間のかかるホストタスクは、`zeFenceHostSynchronize` API 呼び出しで、約 54 秒かかっています。この API 呼び出しにより、ホストはフェンス信号が送信されるか、指定されたタイムアウト期間が経過するまで待機します。このリストには、各タスクの合計実行時間、タスクの合計数、および各タスクの平均時間が表示されます。

詳細



Grouping: Task Domain / Task Type / Function / Call Stack

Task Domain / Task Type / Function / Call ...	Task Time ▼	Task Count	Average Task Time	Module	Function (Full)	Source File	Start Address
Level-Zero API	46.777s	100,027	0.000s				0
zeFenceHostSynchronize	43.785s	50,001	0.001s				0
zeCommandQueueExecuteComman	2.600s	50,001	0.000s				0
zeCommandQueueCreate	0.369s	1	0.369s				0
zeMemAllocHost	0.020s	16	0.001s				0
zeCommandListCreate	0.002s	8	0.000s				

図 3: NPU タスクとホストタスクの **[Bottom-Up (ボトムアップ)]** ビュー

次のステップでは、各タスクをより詳しく調査するため、**[Bottom-Up (ボトムアップ)]** ビューに移動します。ここでは、実行タイムラインと、グループを指定して関数とタスクの **[Bottom-Up (ボトムアップ)]** ビューを確認できます。

NPU タスクとホストタスクを理解するには、Task Type/Function/Call Stack (タスクタイプ/関数/コールスタック) のグループ化を推奨します。

図 4 (タイムライン・ビュー) は、CPU と NPU で実行されているスレッドのリストを示しています。また、タイムライン全体にわたって、対応する NPU Utilization (NPU 利用率) と NPU Bandwidth (NPU 帯域幅) も確認できます。ここで NPU がアクティブだった領域を選択すると、推論の実行時間が約 54 秒であることが分かります。

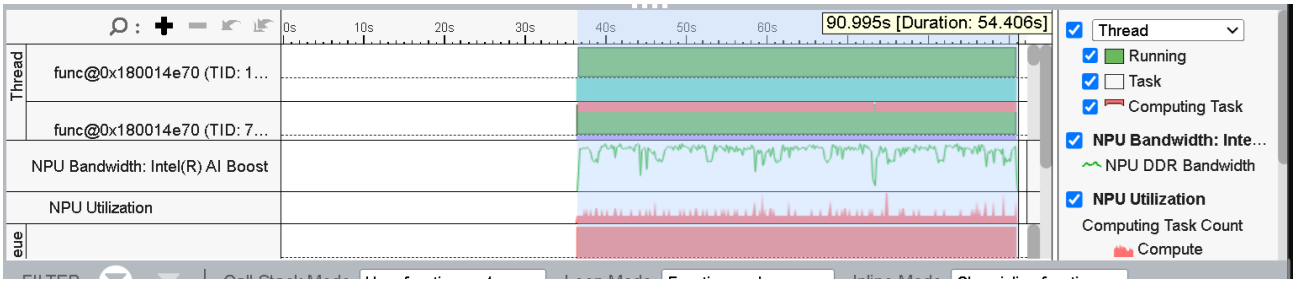


図 4: アプリケーション実行の **[Timeline (タイムライン)]** ビュー

残りの実行時間 (最初の 35 秒) は、ほかの CPU タスク (統計収集、高速バイアス補正) に費やされます。**[Thread (スレッド)]** セクションを展開すると、CPU アクティビティーの詳細を確認できます。

図 5 に示すように、メモリー割り当てやキュー作成などの CPU タスクは、推論開始直前に実行されます。

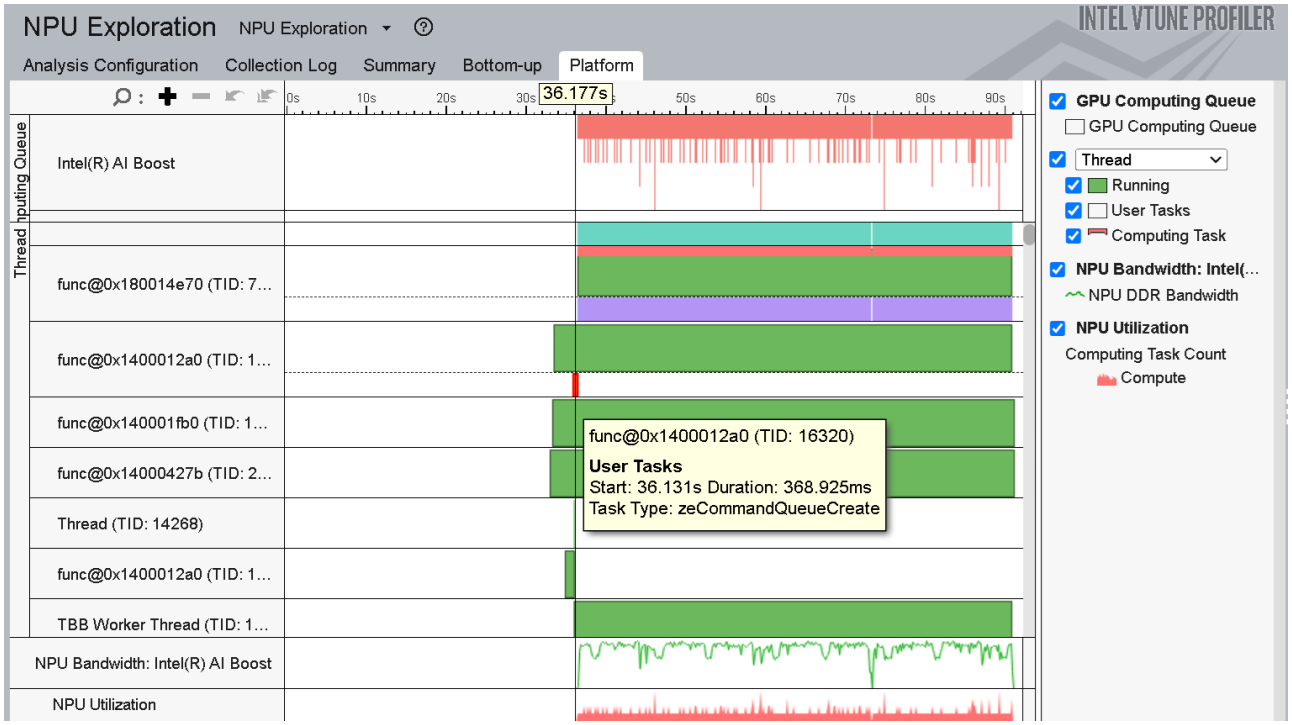


図 5: [Bottom-up (ボトムアップ)] ビューポイントの **[Thread (スレッド)]** セクションを展開した状態

NPU 利用率または NPU 帯域幅に低下が見られる場合は、そのタイムスライスを拡大表示して対応する CPU タスクと NPU タスクを確認し、パフォーマンス低下の原因となっている API またはタスクを把握できます。

NPU 利用率または NPU 帯域幅に低下が見られる場合は、ホスト側のタスク (紫色でマークされています) を確認します。より深く理解するには、対応するスレッドを展開し、タスクにマウスポインターを合わせると詳細が表示されます。

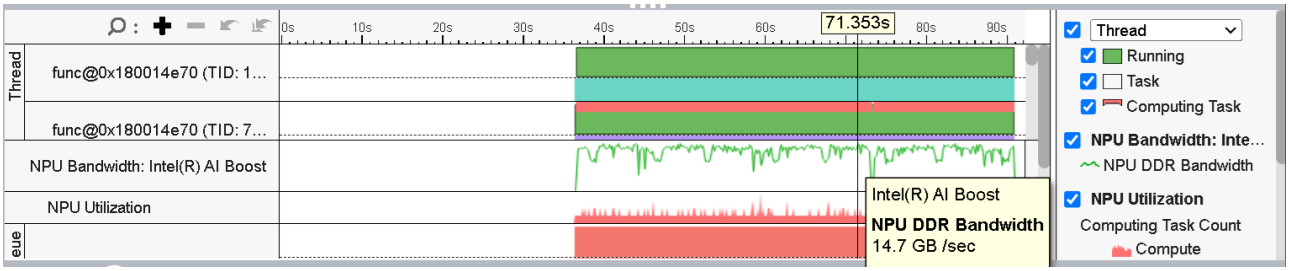


図 6: NPU DDR 帯域幅

タイムライン・ビューでは、実行時間全体にわたる NPU DDR 帯域幅の利用率も確認できます。図 6 は、NPU DDR 帯域幅のピークが約 14GB/秒であることを示しています。

一般的な NPU 利用率のボトルネック

インテル® VTune™ プロファイラーで確認されたボトルネック:

1. DDR から NPU へのデータ転送量の増加により、データ転送のオーバーヘッドが増加
2. NPU メモリーへの負荷が高い
3. ホスト側の同期にかなりの時間がかかる
4. NPU 利用率と NPU 帯域幅利用率の低下

推奨事項

可能な場合は、低精度データを使用して、データ転送量とメモリー・サブシステムへの負荷を軽減します。低精度データは、ホスト-デバイス間の通信におけるレイテンシーと同期オーバーヘッドを軽減します。

4. 結果に基づいてワークロードを最適化する

このステップでは、インテル® VTune™ プロファイラーで検出したボトルネックに応じて最適化手法を適用します。Neural Network Compression Framework (NNCF) のトレーニング後の量子化アルゴリズムを使用してモデルを量子化します。量子化の目的は、上記の推奨事項に従って、メモリー・サブシステムへの負荷を軽減することです。

コード最適化に伴う変更:

- nncf.quantize 関数とキャリブレーション・データセットを使用して FP32 モデルを量子化。
- 量子化後に量子化モデルを保存。
- NPU デバイスで量子化モデルのベンチマークを実施。

```
import re
import subprocess
from pathlib import Path
from typing import List

import numpy as np
import openvino as ov
import torch
from fastdownload import FastDownload
from rich.progress import track
```

```
from sklearn.metrics import accuracy_score
from torchvision import datasets
from torchvision import transforms

import nncf

ROOT = Path(__file__).parent.resolve()
DATASET_PATH = Path().home() / ".cache" / "nncf" / "datasets"
MODEL_PATH = Path().home() / ".cache" / "nncf" / "models"
MODEL_URL =
"https://huggingface.co/alexsu52/mobilenet_v2_imagenette/resolve/main/openvino_model.tgz"
DATASET_URL = "https://s3.amazonaws.com/fast-ai-imageclas/imagenette2-320.tgz"
DATASET_CLASSES = 10


def download(url: str, path: Path) -> Path:
    downloader = FastDownload(base=path.resolve(), archive="downloaded", data="extracted")
    return downloader.get(url)


def validate(model: ov.Model, val_loader: torch.utils.data.DataLoader) -> float:
    predictions = []
    references = []

    compiled_model = ov.compile_model(model, device_name="NPU")
    output = compiled_model.outputs[0]

    for images, target in track(val_loader, description="Validating"):
        pred = compiled_model(images)[output]
        predictions.append(np.argmax(pred, axis=1))
        references.append(target)

    predictions = np.concatenate(predictions, axis=0)
    references = np.concatenate(references, axis=0)
    return accuracy_score(predictions, references)


def run_benchmark(model_path: Path, shape: List[int]) -> float:
    cmd = ["benchmark_app", "-m", model_path.as_posix(), "-d", "GPU", "-api", "async", "-
niter", "50000", "-shape", str(shape)]
    cmd_output = subprocess.check_output(cmd, text=True)  # nosec
    print(*cmd_output.splitlines()[-8:], sep="\n")
    match = re.search(r"Throughput\: (.+?) FPS", cmd_output)
    return float(match.group(1))


def get_model_size(ir_path: Path, m_type: str = "Mb") -> float:
    xml_size = ir_path.stat().st_size
    bin_size = ir_path.with_suffix(".bin").stat().st_size
    for t in ["bytes", "Kb", "Mb"]:
        if m_type == t:
            break
        xml_size /= 1024
        bin_size /= 1024
    model_size = xml_size + bin_size
    print(f"Model graph (xml):   {xml_size:.3f} Mb")
    print(f"Model weights (bin): {bin_size:.3f} Mb")
    print(f"Model size:          {model_size:.3f} Mb")
    return model_size


# ##############################################################################
# Create an OpenVINO model and dataset

dataset_path = download(DATASET_URL, DATASET_PATH)

normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
val_dataset = datasets.ImageFolder(
```

```
    root=dataset_path / "val",
    transform=transforms.Compose(
        [
            transforms.Resize(256),
            transforms.CenterCrop(224),
            transforms.ToTensor(),
            normalize,
        ]
    ),
)
val_data_loader = torch.utils.data.DataLoader(val_dataset, batch_size=1, shuffle=False)

path_to_model = download(MODEL_URL, MODEL_PATH)
ov_model = ov.Core().read_model(path_to_model / "mobilenet_v2_fp32.xml")

def transform_fn(data_item):
    images, _ = data_item
    return images


calibration_dataset = nncf.Dataset(val_data_loader, transform_fn)
ov_quantized_model = nncf.quantize(ov_model, calibration_dataset)

###############################################################################
# Benchmark performance, calculate compression rate and validate accuracy

int8_ir_path = ROOT / "mobilenet_v2_int8.xml"
ov.save_model(ov_quantized_model, int8_ir_path)
print(f"[2/7] Save INT8 model: {int8_ir_path}")
int8_model_size = get_model_size(int8_ir_path)

print("[4/7] Benchmark INT8 model:")
int8_fps = run_benchmark(int8_ir_path, shape=[1, 3, 224, 224])
```

最適化を適用後、インテル® VTune™ プロファイラーを使用して以下のメトリックを検証します。

1. DDR から NPU へのデータ転送量の変化を把握します。
2. インテル® VTune™ プロファイラーのタイムライン・ビューから推論時間を観察します。
3. NPU メモリーのピーク帯域幅を確認します。

観測された実行速度の向上

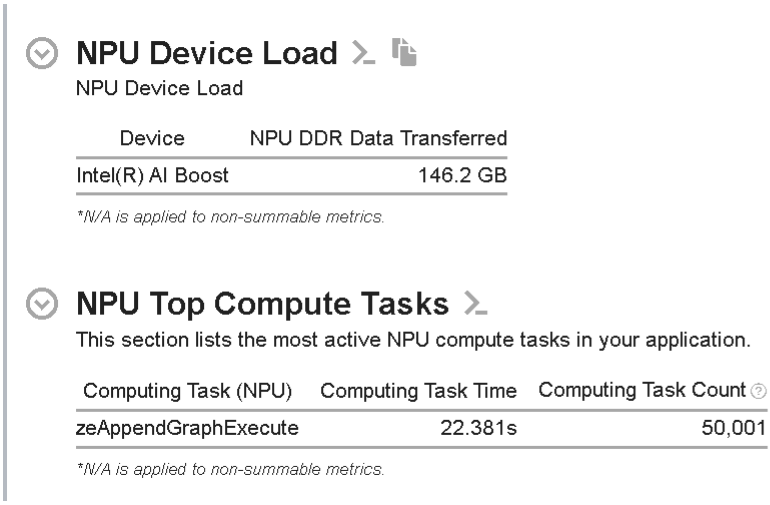


図 7: 量子化モデルの NPU Device Load (NPU デバイスロード) と NPU Top Compute Tasks (NPU の上位計算タスク)

図 7 に示すように、NPU デバイスロードは約 680GB から約 146GB に減少しており、これは NPU DDR データ転送量が約 78.5% 改善されたことを示しています。データ転送オーバーヘッドの改善により、NPU での推論時間が短縮されるはずです。

また、**[NPU Top Compute Tasks (NPU の上位計算タスク)]** セクションでは、最も時間のかかるタスク `zeAppendGraphExecute` が約 22 秒にまで短縮され、37.14% の改善が見られました。

Top Tasks

This section lists the most active tasks in your application.

Task Type	Task Time	Task Count	Average Task Time
zeFenceHostSynchronize	28.173s	50,001	0.001s
zeCommandQueueExecuteCommandLists	2.242s	50,001	0.000s
zeCommandQueueCreate	0.383s	1	0.383s
zeMemAllocHost	0.010s	16	0.001s
zeCommandListCreate	0.003s	8	0.000s

**N/A is applied to non-summable metrics.*

図 8: 量子化後のホスト側の上位タスク

ホスト側の上位タスクにも大きな違いが見られます。例えば、`zeFenceHostSynchronize` と `zeCommandQueueExecuteCOmmandLists` の実行時間は、量子化されていないモデルの実装と比較して大幅に短縮されています。

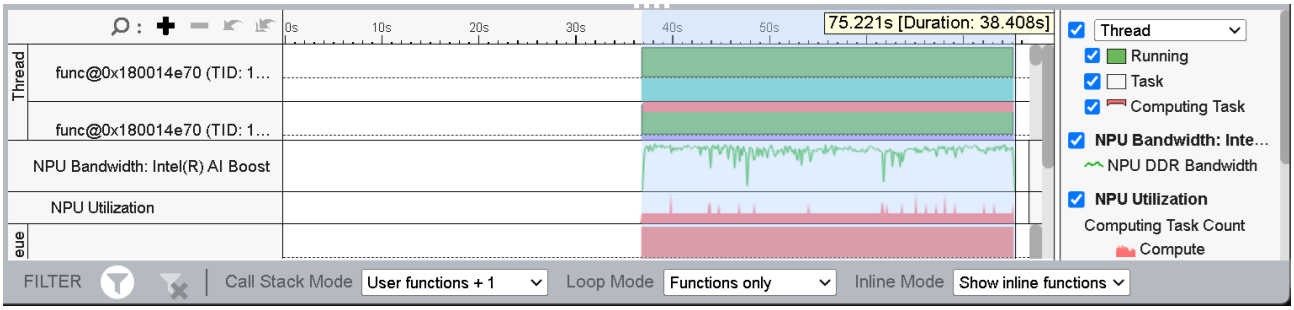


図 9: 量子化モデルの推論時間 (タイムライン・ビュー)

図 9 に示すように、推論の実行時間は約 38 秒となり、最適化前の実装から 30% も高速化しました。NPU がアクティブな間のタイムラインを選択すると、推論時間を確認できます。

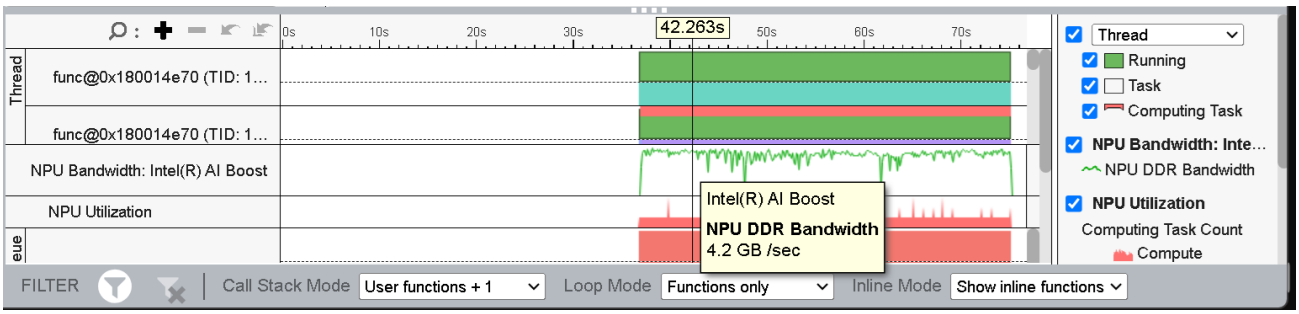


図 10: 量子化モデルの NPU 帯域幅

図 10 に示すように、ピーク時の NPU DDR 帯域幅は約 14GB/秒から約 4GB/秒に低下しています。これにより、NPU メモリー・アクティビティーと DDR から NPU へのデータ転送も減少し、全体的な推論パフォーマンスが大幅に向上します。

実際に試してみる

これらの手順を実行して独自のアプリケーションに適用するには、以下の手順に従ってください。

- この記事の冒頭で説明したように、環境をセットアップしてください。
- こちらのサンプル (英語) を実行するか、独自の開発プロジェクトを使用します。
- インテル® VTune™ プロファイラーを使用してアプリケーションをプロファイルし、ボトルネックを検出して修正します。

ソフトウェアを入手

インテル® VTune™ プロファイラーは無料で、インテル® oneAPI ベース・ツールキットの一部として、またはスタンドアロン版 (英語) としてインストールできます。

ツールキット・セレクター (英語) を使用して、フルキットまたは特定のユースケースに必要なコンポーネントのみを含むより小さなサイズの新しいサブバンドルをインストールすることで、ソフトウェアのセットアップを効率化できます。インテル® C++ エッセンシャルズ、インテル® Fortan エッセンシャルズ、インテル® ディープラーニング・エッセンシャルズを利用して、時間を節約し、手間を減らし、プロジェクトに最適なツールを入手できます。

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。