今すぐ Python* を高速化

アクセラレーター・アーキテクチャーのカンブリア爆発に備える

James Reinders インテル コーポレーション oneAPI エバンジェリスト兼 The Parallel Universe 名誉編集長

Python* は、優れた汎用性とパフォーマンスで、常に多くの人を驚かせています。私は生粋の C と Fortran のプログラマーですが、ハイパフォーマンスを実現するため C++ もかなり使いこなすことができます。Python* もハイパフォーマンスを実現できますが、先に挙げた言語とは一線を画す利便性を備えているため、私は Python* のファンでもあります。

高度に最適化された主要なライブラリーと、プリコンパイルされないコードに対する（実行時の）JIT コンパイルサポートにより、Python* はハイパフォーマンスを実現します。しかし、Python* コードは、大きなデータセットや複雑なアルゴリズムでは遅くなる傾向があります。本記事では、以下の項目について説明します。

1. 「ヘテロジニアスな未来」を考えることの重要性
2. オープン・ソリューションで解決すべき 2 つの重要な課題
3. 利用可能な CPU パフォーマンスを効率良く活用する並列実行
4. アクセラレーターを使用してパフォーマンスをさらに向上する

3 番目の並列実行だけで 12 倍高速化でき、4 番目のアクセラレーターの使用によりさらに高速化することができます。これらの手法は、Python* プログラマーがパフォーマンスを向上させたいときに非常に有効であり、簡単に取り入れることができます。ここで紹介する手法は、結果が出るまでにあまり時間がかかりません。

「ヘテロジニアスな未来」を考える

Python* コードを高速に実行したいだけなら、ヘテロジニアスを理解することは重要ではありませんが、現在コンピューティングで起きている大きな変化を知っておく価値はあります。コンピューターは年々高速化しています。当初は、より巧妙で複雑なアーキテクチャーが、パフォーマンスの向上を牽引していました。1989 年から 2006 年にかけては、クロック周波数の向上が主な原動力となりました。そして、2006 年に突然、クロック周波数の向上が鈍化し、パフォーマンスを向上するには再びアーキテクチャーの変更が必要になったのです。

マルチコア・プロセッサーは、プロセッサー内の（同種の）コアの数を増やすことで、より高いパフォーマンスを提供するものでした。クロック周波数の向上とは異なり、マルチコアのパフォーマンスを引き出すには、ソフトウェアを変更する必要がありました。Herb Sutter 氏の名著「The Free Lunch Is Over（フリーランチは終わった）」（英語）は、並行処理の必要性を強調しました。並列化は必要でしたが、この変化はソフトウェア開発者に困難な課題をもたらしました。

次に、CPU の計算処理を専用のデバイスで補強するアクセラレーターが登場しました。その中で最も成功したのが GPU です。GPU は元々、グラフィックス処理をコンピューターのディスプレイにオフロードするために導入されました。GPU の計算能力を利用するいくつかのプログラミング・モデルが登場しましたが、結果をディスプレイに送るのではなく、CPU 上で動作するプログラムに送り返すものでした。その中で最も成功したモデルは、NVIDIA* GPU 向けの CUDA* です。現在では、同じシステム内の（ヘテロジニアスな）プロセッサーの処理能力は同等ではありません。しかし、一般的なプログラミング言語は、単一の計算デバイスを前提としているため、コードの一部を別の計算デバイスで実行する場合、「オフロード」という用語が使われます。

数年前、業界のレジェンドである John Hennessey 氏と David Patterson 氏が、「A New Golden Age for Computer Architecture（コンピューター・アーキテクチャーの新しい黄金時代）」に入ったと発表しました。ヘテロジニアス・コンピューティングは、多くのドメイン専用のプロセッサーの登場により、爆発的に増加しています。成功するものもあれば失敗するものもありますが、もはやすべての計算を 1 つのデバイスで行う時代は終わり、コンピューティングは永遠に変化し続けるでしょう。

2 つの重要な課題を 1 つの優れたソリューションで解決

CUDA* は人気がありますが、その対象は NVIDIA* GPU に限定されています。複数のベンダーから登場する新しいアクセラレーター・アーキテクチャーに対応するには、オープンなソリューションが必要です。ヘテロジニアス・プラットフォームで実行されるプログラムは、実行時に利用可能なデバイスを検出する必要があります。また、これらのデバイスに計算をオフロードする仕組みも必要です。

CUDA* は、NVIDIA* GPU のみが利用可能であると仮定し、デバイスの検出を行いません。Python* ユーザーは、NVIDIA* CUDA* や AMD* ROCm* で GPU を活用するため、CuPy（英語）を選択できます。CuPy は堅実な選択肢ですが、CPU パフォーマンスを向上したり、ほかのベンダーやアーキテクチャー向けに汎用化することはできません。複数のベンダーに移植可能で、新しいハードウェア・イノベーションをサポートできるプログラミング・ソリューションがあれば、もっと良い結果が得られるでしょう。アクセラレーターへのオフロードを検討する前に、ホスト CPU を最大限に活用していることを確認します。並列性を引き出す方法とコンパイルされたコードを理解すれば、アクセラレーターでも同様の並列処理を利用できます。

Numba（英語）は、Anaconda 社によって開発された、オープンソースの Python* 用 NumPy* 対応最適化（JIT）コンパイラーです。LLVM コンパイラーを使って Python* バイトコードからマシンコードを生成します。Numba は、多くの NumPy* 関数を含む、数値計算に特化した Python* の大部分をコンパイルできます。また、ループの自動並列化、GPU アクセラレーション・コードの生成、ユニバーサル関数（ufuncs）と C コールバックの生成をサポートしています。

Numba の自動並列化（英語）はインテルが貢献したもので、`@numba.jit` で `parallel=True` オプションを設定すると有効になります。自動並列化は、コンパイルされた関数内のデータ並列コード領域を分析し、並列実行のスケジュールを作成します。Numba は、次の 2 種類の処理を自動並列化できます。

1. NumPy* の配列式、ufuncs、リダクション関数などの暗黙的なデータ並列領域
2. numba.prange 式で指定される明示的なデータ並列ループ

例えば、次の単純な Python* ループについて考えてみます。

```
def f1(a,b,c,N):
    for i in range(N):
        c[i] = a[i] + b[i]
```

シリアル領域（`range`）を並列領域（`prange`）に変更し、`njit` ディレクティブ（`njit` は Numba JIT、並列バージョンをコンパイルする）を追加することで、明示的に並列化できます。

```
@njit(parallel=True)
def add(a,b,c,N):
    for i in prange(N):
        c[i] = a[i] + b[i]
```

このコードを実行したところ、実行時間は 24.3 秒から 1.9 秒に改善されました。結果はシステムによって異なります。このコードを試すには、oneAPI-samples リポジトリーをクローンし（`git clone https://github.com/oneapi-src/oneAPI-samples`（英語））、`AI-and-Analytics/Jupyter/Numba_DPPY_Essentials_training/Welcome.ipynb` ノートブックを開きます。インテル® DevCloud for oneAPI（英語）の無料のアカウントを取得すると、簡単に試すことができます。

アクセラレーターを使用してパフォーマンスをさらに向上

アクセラレーターは、アプリケーションにオフロードのオーバーヘッドに見合うワークがある場合に、非常に効果的です。最初のステップは、オフロードできるように計算（カーネル）を選択してコンパイルすることです。前の例を拡張して、オフロードカーネルを指定するため、Numba データ並列拡張（numba-dpex（英語））を使用します（詳細は、Jupyter* Notebook トレーニング（英語）を参照してください）。

```
@dppy.kernel
def add(a, b, c):
    i = dppy.get_global_id(0)
    c[i] = a[i] + b[i]
```

カーネルコードのコンパイルと並列化を行います。前述の例では、`@njit` を使用して CPU 上で実行しましたが、今回はデバイスにオフロードします。カーネルコードは、実行時にランタイムがデバイスにマップする中間言語（SPIR-V*（英語））にコンパイルされます。これにより、ベンダーに依存しないアクセラレーター・オフロードのソリューションが得られます。

カーネルへの配列引数は、プログラミング・ニーズに応じて、NumPy* 配列または USM（統合共有メモリー）配列（USM に明示的に配置される配列型）のいずれかを選択できます。この選択は、データの設定やカーネルの呼び出し方法に影響します。

次に、オープンソースのデータ並列制御（dpctl：SYCL* 向けの C および Python* バインディング）を使用して、SYCL* というオープンなマルチベンダー、マルチアーキテクチャー・プログラミングの C++ ソリューションを利用します（詳細は、GitHub* のドキュメント（英語）と「XPU プログラミング向けの SYCL と Python* のインターフェイス」（英語）を参照してください）。これにより、Python* プログラムは SYCL* デバイス、キュー、メモリーリソースにアクセスし、Python* の配列 / テンソル操作を実行できるようになります。これは、ソリューションの再考案を回避し、学習量を減らし、高いレベルの互換性の確保につながります。

デバイスへの接続は簡単です。次のコードを追加するだけです。

```
device = dpctl.select_default_device()
print("Using device ...")
device.print_device_info()
```

プログラムを変更せずにデバイスの選択を制御したい場合は、環境変数 `SYCL_DEVICE_FILTER` でデフォルトのデバイスを設定できます。`dpctl` ライブラリーは、ハードウェアのプロパティーに基づいて利用可能なデバイスをレビューし選択する、プログラムによる制御もサポートしています。

カーネルは、数行の Python* コードによりデバイス上で起動（オフロードして実行）できます。

```
with dpctl.device_context(device):
    dpar_add[global_size,dppy.DEFAULT_LOCAL_SIZE](a,b,c)
```

`device_context` を使用することで、ランタイムによって必要なデータコピー（この例ではデータは標準 NumPy* 配列のままです）が行われます。`dpctl` ライブラリーは、明示的にデバイスの USM メモリーを割り当て、管理する機能もサポートしています。これは、高度な最適化を行う際には重要な機能ですが、ランタイムが標準 NumPy* 配列を処理する手軽さに勝るものはありません。

非同期と同期

Python* のコーディング・スタイルは、上に示した同期メカニズムによって簡単にサポートされます。非同期の機能とその利点（データ移動とカーネル呼び出しの待ち時間を減らすまたは隠す）は、Python* コードを少し変更すれば利用できます。非同期実行の詳細については、dpctl gemv example（英語）にあるサンプルコードを参照してください。

CuPy

CuPy（英語）は NumPy* の大部分を再実装したものです。CuPy 配列ライブラリーは、既存の NumPy*/SciPy* コードを NVIDIA* CUDA* や AMD* ROCm* プラットフォームで実行するドロップイン置換として機能します。しかし、新しいプラットフォーム向けに CuPy を再実装するには、膨大なプログラミング作業が必要であり、同じ Python* プログラムでマルチベンダーをサポートする大きな障壁となり、前述の 2 つの重要な課題に対応できません。CuPy では CUDA* 対応の GPU デバイスが必要です。メモリーについては、`cudaMalloc` の呼び出し回数を減らすため、メモリープーリングを自動的に実行しますが、直接制御することはほとんどできません。カーネルのオフロードでは、デバイスの選択を制御できず、CUDA* 対応 GPU が利用できない場合は失敗します。ヘテロジニアスの課題に対応した優れたソリューションを使用することは、アプリケーションの移植性を高めます。

scikit-learn*

Python* プログラミングは一般に「compute-follows-data」に適しており、そのルーチンはとてもシンプルです。`dpctl` ライブラリーは、特定のデバイスと接続するテンソル配列型をサポートしています。このプログラムでは、データをデバイステンソルにキャストすると［例：`dpctl.tensor.asarray(data, device="gpu:0")`］、データがデバイスと関連付けられて、デバイス上に配置されます。このデバイステンソルを認識する scikit-learn* のパッチを適用したバージョン（英語）を使用すると、このテンソルを含むパッチを適用した scikit-learn* のメソッド（英語）は自動的にデバイス上で計算されます。

これは Python* の動的型付けの優れた使い方で、データがどこにあるかを検出し、データが存在する場所で計算を行うように指示します。Python* コードはほとんど変わらず、唯一の変更は、テンソルをデバイステンソルに再キャストするところです。これまでユーザーから寄せられたフィードバックから、compute-follows-data メソッドは Python* ユーザーに最も人気のモデルになると予想されます。

オープン、マルチベンダー、マルチアーキテクチャー — ともに学ぶ

Python* は、ハードウェアの多様性を受け入れ、差し迫ったアクセラレーターのカンブリア爆発を利用する道具となりえます。Numba データ並列 Python*、`dpctl`、compute-follows-data（パッチを適用した scikit-learn*）の組み合わせは、ベンダーやアーキテクチャーにとらわれないため、検討する価値があります。

Numba は NumPy* に素晴らしいサポートを提供していますが、将来的には SciPy* やほかの Python* のニーズへの対応を検討できるでしょう。Python* の配列 API の断片化により、CPU 以外のデバイスでワークロードを共有したいという要望から、Python* 配列 API の標準化（要約はこちら（英語））への関心が高まっています。標準配列 API は、Numba や `dpctl` などの取り組みの範囲と影響を拡大するのに大いに役立っています。NumPy* と CuPy は配列 API を採用しており、`dpctl` と PyTorch*（英語）は採用に向けて取り組んでいます。より多くのライブラリーに採用が拡大すれば、ヘテロジニアス・コンピューティング（多種多様なアクセラレーター）のサポートが容易になるでしょう。

複数のスレッドや非同期タスクを使用するより高度な Python* コードでは、`dpctl.device_context` を使用するだけでは十分ではありません（GitHub* にある問題（英語）を参照してください）。少なくとも、より複雑なスレッド化された Python* コードでは、compute-follows-data を採用することをお勧めします。これは、`device_context` スタイルのプログラミングよりも優先されるオプションになる可能性があります。

Python* を高速化するため、皆で貢献し、一緒に改善に取り組むことができます。すべてがオープンソースであり、非常にうまく機能しています。

関連情報

実際に試してみることで多くのことを学べます。いくつかの役立つオンラインリソースを紹介します。

Data Parallel Essentials for Python（Python* のデータ並列の基本）(英語) — Numba と dpctl の概念について説明した 90 分のビデオ

Losing your Loops Fast Numerical Computing with NumPy（NumPy* による高速な数値計算でループを排除する）(英語) —『Python Data Science Handbook（Python* データ・サイエンス・ハンドブック）』（英語）の著者である Jake VanderPlas 氏による NumPy* の効果的な使用に関するビデオ

本記事で紹介したヘテロジニアスの Python* 機能はすべてオープンソースで、インテル® oneAPI ベース・ツールキットとインテル® AI アナリティクス・ツールキットに含まれています。SYCL* 対応の NumPy* は GitHub*（英語）でホストされています。カーネル・プログラミングと自動オフロード機能の Numba コンパイラー拡張も GitHub* でホストされています。オープンソースのデータ並列制御（dpctl : SYCL* 向けの C および Python* バインディング）には、GitHub* のドキュメント（英語）と、論文「XPU プログラミング向けの SYCL* と Python* のインターフェイス」があります。これにより、Python* プログラムは SYCL* デバイス、キュー、メモリーにアクセスし、Python* の配列 / テンソル操作を実行できるようになります。

デバイスコードからの非同期エラーを含む例外がサポートされています。非同期エラーは、非同期エラーハンドラー関数によって同期例外として再スローされると、インターセプトされます。この動作は Python* の拡張ジェネレーターにより提供されるもので、コミュニティーのドキュメント Cython（英語）と Pybind11（英語）で詳しく説明されています。