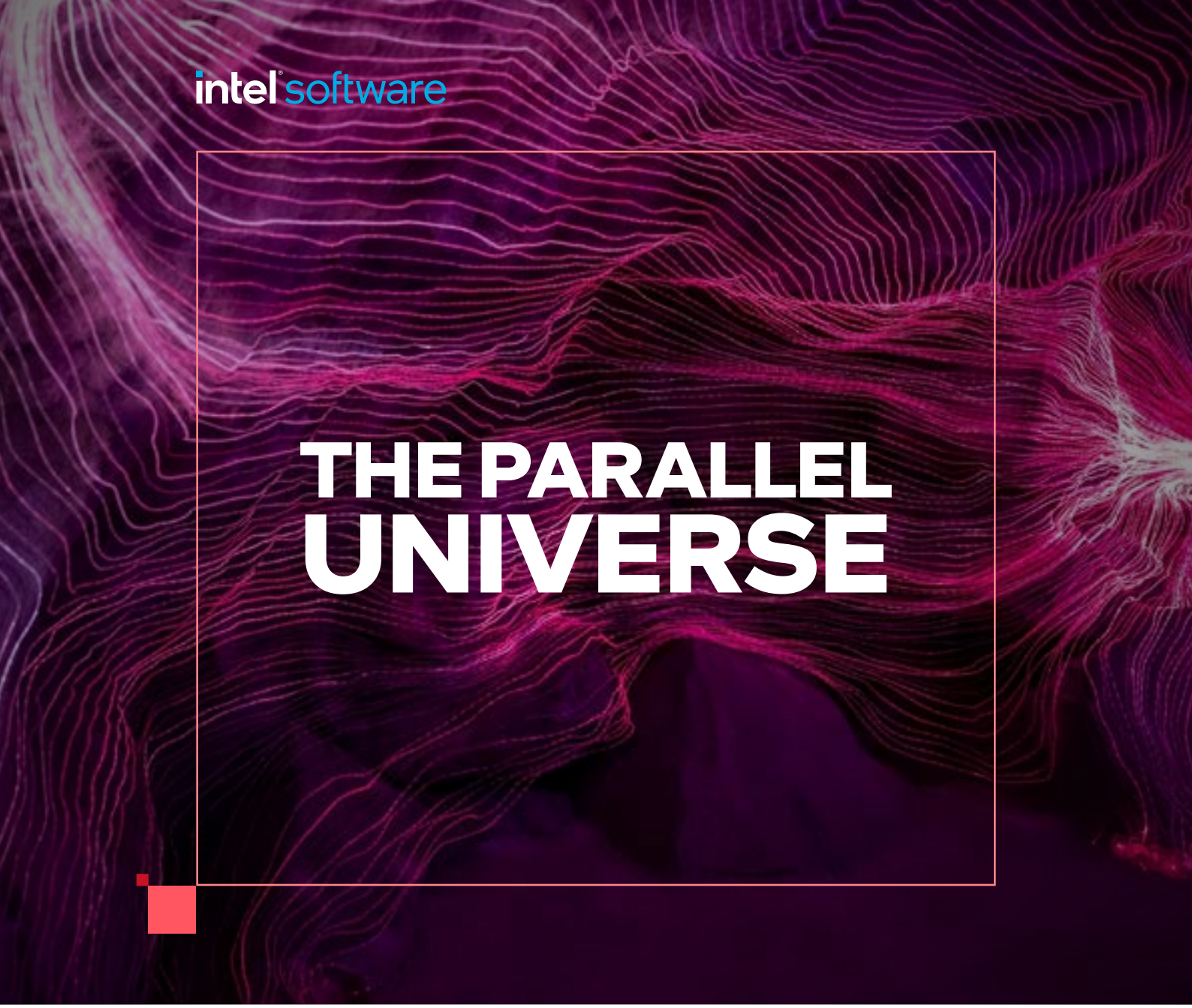


SYCL とインテル® oneMKL による定量金融の高速化

ポータブルな Python* 向けデータ・パラレル・エクステンション：GPU を活用した計算の高速化

PCA と DBSCAN による時系列クラスタリング

Issue
58
2025

目次

編集者からのメッセージ 3

注目記事

SYCL とインテル® oneMKL による定量金融の高速化 5

ポータブルな Python* 向けデータ・パラレル・エクステンション : GPU を活用した計算の高速化 17

PCA と DBSCAN による時系列クラスタリング 22

Transformer ベースの時系列予測器の実装 28

JAX と OpenXLA の実行プロセスと基本ロジック : パート 1 35

JAX と OpenXLA の実行プロセスと基本ロジック : パート 2 45

編集者からのメッセージ

インテル コーポレーション AI/ML 担当テクニカル・プロダクト・マーケティング・マネージャー Susan E. Kahler
テクノロジー業界における革新的なソリューションの推進に尽力しており、PyTorch Foundation マーケティング委員会および Linux Foundation AI & Data Foundation アウトリーチ委員会のメンバーとして、コミュニティーのエンゲージメントとコラボレーションに貢献しています。人間工学の博士号を取得しており、数学的アルゴリズムを用いて人間の学習モデルの分析を行っています。ユーザー中心設計、製品管理、顧客インサイト、運用リスクなど、多様な分野での経歴を持ち、ユーザー体験の向上とテクノロジーの進歩において信頼できる専門家としての地位を確立しています。

この度、『The Parallel Universe』の編集を担当することになりました Susan E. Kahler です。本号の内容について説明する前に、2016年の第 27 号から『The Parallel Universe』の編集者として卓越したリーダーシップを発揮した Henry Gabb 氏に心から感謝の意を表します。Gabb 氏は最近インテルを退職されましたが、私はこの役割を引き継ぐことを光栄に思います。彼のハイパフォーマンス・コンピューティングと並列コンピューティングへの献身は、本誌を充実させただけでなく、この分野の多くの実務者にインスピレーションを与えました。洞察に満ちた論文の執筆や並列プログラミングの発展に向けた共同作業など、彼の計り知れない貢献と、彼が築いた高い基準と揺るぎないサポートに深く感謝します。今後も、本誌がこの基準を維持できるよう、全力を尽くします。

本号では、AI、データサイエンス、ハイパフォーマンス・コンピューティングの最新動向を探る 7 本の記事を紹介します。

注目記事「**SYCL とインテル® oneMKL による定量金融の高速化**」では、Andrey Fedorov と Robert Mueller-Albrecht が、金融市場がギリシャ指標（リスク・パラメーター）を用いた数学モデルにどのように依存しているかについて解説します。また、金融工学が定量金融の数学的理論と計算シミュレーションを組み合わせて、価格、取引、ヘッジ、その他の投資判断を行う仕組みについても示します。

ヘテロジニアス・プラットフォームの普及に伴い、Unified Acceleration Foundation（UXL）は、ベンダーやプラットフォームに依存しないソリューションに注目した、プログラミング・アクセラレーター向けのオープン・スタンダード・ソフトウェア・エコシステムを構築しています。Nikita Grigorian と Oleksandr Pavlyk の記事「**ポータブルな Python* 向けデータ・パラレル・エクステンション：GPU を活用した計算の高速化**」では、UXL を Python に拡張することで、ユーザーがポータブルなデータ並列ネイティブ拡張機能を開発し、同一セッション内でさまざまなベンダーのアクセラレーターで計算できるようにする方法について説明します。

続く 2 つの記事では、Bob Chesebrough が時系列分析に取り組む顧客と協力し、新旧両方の予測とクラスタリング手法を調査した結果を、「**PCA と DBSCAN による時系列クラスタリング**」と「**Transformer ベースの時系列予測の実装**」（Jack Erickson との共著）の 2 つの記事に分けて紹介します。

「**JAX と OpenXLA の実行プロセスと基本ロジック：パート 1**」と「**JAX と OpenXLA の実行プロセスと基本ロジック:パート 2**」では、Zhenming Wang とそのチームが JAX フレームワークと OpenXLA を深く掘り下げ、Python プログラムでこれらのテクノロジーを活用して効率良く実行する方法を 2 部構成で紹介します。

開発者コミュニティーにとって、富士通がインテル® oneAPI データ・アナリティクス・ライブラリー（インテル® oneDAL）の Arm アーキテクチャーへの移植に成功したことは画期的な出来事です。詳細は、Fujitsu Tech Blog の「Developer Story:How We Ported oneDAL on Arm for Accelerated AI Workloads for FUJITSU-MONAKA（開発者ストーリー : FUJITSU-MONAKA の AI ワークロードを高速化するため Arm に oneDAL を移植した方法）」（英語）を参照してください。本号ではこの記事を取り上げていませんが、さまざまな ML アルゴリズムで達成されたパフォーマンス向上をぜひご覧ください。

これらの記事が、皆さんの業務における新しい手法やツールの探求のインスピレーションとなることを願っています。AI とデータサイエンス、コードの現代化、ビジュアル・コンピューティング、データセンターとクラウド・コンピューティング、システムと IoT 開発、インテル® oneAPI ツールキットを利用したヘテロジニアス並列コンピューティング向けのインテル・ソリューションの詳細は、Tech.Decoded（英語）を参照してください。

今後『The Parallel Universe』で取り上げて欲しいトピックがありましたら、LinkedIn（英語）からご連絡ください。

Susan E. Kahler
2025 年 4 月

SYCL とインテル® oneMKL による定量金融の高速化

Andrey Fedorov インテル コーポレーション 数学アルゴリズム・エンジニア
Robert Mueller-Albrecht インテル コーポレーション テクニカル・プロダクト・マーケティング・エンジニア

金融市場では、さまざまな証券のリスクの分析、予測、評価に数学モデルを利用し、大規模データセットに適用しています。一般的な例としては、オプションなどのデリバティブ証券の価格決定や、ポートフォリオ管理におけるリスク管理が挙げられます。金融工学は、定量金融の数学的理論と計算シミュレーションを組み合わせて、価格決定、取引、ヘッジ、その他の投資判断を行います。

これらの計算には以下が使用されます。

- 微分方程式の数値解
- 多数のランダムシードを用いたモンテカルロ・シミュレーションに基づく証券価格決定シミュレーション

株式、株価指数、コモディティー、債券、通貨、転換社債契約、さらにはスワップや先物といったデリバティブなど、多くの金融商品はオプション市場に依存し、その影響を受けます。そのため、金融市場にとって、オプション価格決定の挙動を確実にモデル化し、ほかの資産クラスへの影響を適切に理解することが極めて重要です。

ギリシャ指標（リスク・パラメーター）

ギリシャ指標は、オプションやその他の資産クラスのリスク・プロファイルの評価基準となる、オプションに関連する一連のリスク・パラメーターであり、それぞれのリスクはギリシャ記号で表されます。追跡されるリスクは多岐にわたり、最も一般的なものは以下のとおりです。

- シータ（Theta）: 時間経過に対するオプション価格の減衰、または感応度
- ロー（Rho）: 金利変動に対するオプション価格の感応度
- デルタ（Delta）: 原資産価格の変動に対するオプション価格の感応度
- ガンマ（Gamma）: オプションのデルタと原資産価格の変動率に対するオプション価格の 2 次感応度
- ベガ（ニュー）（Vega（nu））: 原資産価格の変動率に対するオプション価格の感応度

ギリシャ指標の微分方程式の数値解は、平方根、指数関数、対数、正規乱数生成、相関乱数生成、パス生成など、さまざまな数学関数に依存します。

インテル® oneAPI マス・カーネル・ライブラリー（インテル® oneMKL）は、C、Fortran、SYCL API を用いたこれらの計算をサポートします。マルチアーキテクチャー GPU アクセラレーター・オフロードを活用するオープンソース、オープン・スタンダード・ベースの実装には、Unified Acceleration Foundations（UXL）のオープンソースの oneAPI マス・ライブラリー（oneMath）（英語）プロジェクトの SYCL API を使用できます。

シミュレーションと微分方程式の数値解を組み合わせた計算集約的な反復プロセスにより、金融資産のリスクモデリングは、SYCL のような高度な並列プログラミング・モデルの理想的な候補となります。

市場評価に用いられるさまざまな予測モデルと、インテル® データセンター GPU マックス・シリーズや最新世代のインテル® Core™ Ultra モバイル・プロセッサーの統合 GPU などの、高度に並列化された計算エンジン上で最適化された乱数生成器（RNG または RAND）を実行するメリットについて詳しく見ていきましょう。

金融における乱数生成とシミュレーション

現代の定量金融において最も一般的なシミュレーション・モデルは以下のとおりです。

1. 二項式オプション価格モデル（BOPM）
2. ブラック - ショールズ・モデル（BSM）
3. ヘストンモデル
4. ヨーロピアン・モンテカルロ・オプション・モデル（EMC）
5. アメリカン・モンテカルロ・オプション・モデル（AMC）

それぞれに目的があります。

1. 二項式オプション価格モデル（BOPM）

このモデルは、一般化可能な数値手法を用いてオプションを評価します。このモデルは、時間の経過とともに変化する価格の「離散時間」（格子ベース）決定木を使用します。このアプローチは、オプションが満期日前であっても、いつでも権利行使できる場合に有効です。投資リスクは、将来の任意の時点において計算する必要があります。BOPM は、特定の時点ではなく、時間経過に伴う投資の変動に基づいているため、米国株式市場で取引されるオプションでよく使用されています。

二項式オプション価格決定は、ブラック - ショールズなどの他の代替手法よりも計算時間が長くなります。さらに、市場条件パラメーターの数が多い場合にはうまく機能しません。しかし、依存関係が単純なオプションでは、特に長期投資において正確な結果が得られるため頻繁に使用されます。

2. ブラック - ショールズ・モデル（BSM）

ブラック - ショールズは、株式や先物契約の価格が、一定のドリフトとボラティリティーを伴うランダムウォークに従う対数正規分布に従うと仮定します。この仮定により、ブラック - ショールズはオプション契約の価格決定に最適なモデルであると一般的に考えられています。ブラック - ショールズ方程式には、ボラティリティー、原資産価格、オプションの権利行使価格、オプションの満期までの期間、無リスク金利、そしてオプションの種類（コールまたはプット）の 6 つの変数が必要です。注意点としては、実際の市場では、BSM リスク分布予測よりも、高リスクの下落が頻繁に発生する傾向があることです。

3. ヘストンモデル

ヘストンモデルは、金融資産のボラティリティーの経時的変化を含めることで、BSM の欠点に対処します。このモデルでは、資産評価とボラティリティーは一定でも決定論的でもなく、確率的であると仮定します。ヘストンモデルの核となる偏微分方程式は、入力パラメーターを介した多数の連続的ランダムウォークのモンテカルロ・シミュレーションによって計算されます。ヘストンモデルは偏微分方程式であり、正確に解くことはできません。このモデルをオプションの価格決定に使用するには、数学的な近似との併用が必要です。このアプローチのアメリカンオプションとヨーロピアン・オプションのモデルへの適用については、次のセクションで説明します。

4. ヨーロピアン・モンテカルロ・オプション・モデル（EMC）

モンテカルロ・シミュレーションは、さまざまな投資戦略の結果の特性を決定するために広く使用されている、繰り返しランダム・サンプリングに基づく手法です。ヨーロピアン・オプションでは、オプションは満期にのみ権利行使され、価格決定は、権利行使時までの固定期間に基づく、単純な金融ベンチマークです。

S_t を、次の式で表される確率過程に従う、特定の時点 t における株価とします。

$$dS_t = \mu S_t dt + \sigma S_t dW_t$$

ここで、μ はドリフト、σ はボラティリティーで、定数と仮定し、$W = (W_t)$ とします。$t \geq 0$ はランダム・サンプリング分布を表します。dt は時間ステップです。

$0 \leq t \leq T$ で定義されるヨーロピアン・オプションの価格 $V(t,S_t)$ は、原株の価格 S_t に依存します。オプションは $t = 0$ に発行され、満期 $t = T$ に権利行使されます。ヨーロピアン・オプションの場合、満期のオプションの価格 $V(T,S_T)$ は次のように定義されます。

コールオプション：

$$V(T,S_T) = max(S_T - X, 0)$$

プットオプション：

$$V(T,S_T) = max(X - S_T, 0)$$

ここで、X は権利行使価格です。問題は、価格 $V(0,S_0)$ を推定することです。

この問題に対するモンテカルロ・アプローチは、S_T の n 通りの実現可能性をシミュレーションし、$V(T,S_T)$ を平均化し、その平均を係数 e^{-rt} で割り引いて、オプションの現在の価格 $V(0,S_0)$ を算出します。S_T は対数正規分布に従います。

$$S_T = S_0 e^{(r-\sigma^2/2)T+\sigma\sqrt{T}\xi}$$

ここで、ξ は標準正規分布の確率変数です。

5. アメリカン・モンテカルロ・オプション・モデル（AMC）

アメリカン・オプション・モデルは、オプションの権利行使日が固定されていないため、やや複雑です。このオプションは、満期時または満期前にいつでも権利行使できます。これに対応するため、最小二乗モンテカルロ法を採用し、次の 2 つのステップを繰り返します。

- まず、後方帰納法を実行し、すべてのタイムステップのすべての状態に再帰的に値を割り当てます。この値は、その時点のオプション価格の市場価格に対する最小二乗回帰として定義されます。この回帰のオプション価格は、(市場価格に依存する) 権利行使の可能性の値と、権利行使可能性のタイムステップ値の合計として定義されます。
- 次に、すべてのタイムステップのすべての状態を評価する際に、価格パスとその結果得られる状態の値に基づいて、各ステップでオプション行使に関する最適な決定を下すことでオプションの価格を計算します。この 2 番目のステップは、手順に確率的効果を加えるため、複数の価格パスで行うことができます。

アメリカン・モンテカルロ・オプション・モデルが、問題の並列実行パスの計算量を増やすことは明らかです。

計算負荷の高いタスクにおけるインテル® oneMKL SYCL API と GPU オフロードの利点を説明するため、これらの異なる手法の簡略化されたリファレンス・サンプル・ベンチマーク実装を見てみましょう。

インテル® oneMKL を用いた異なるオプション予測手法の実装とベンチマーク

金融オプションモデル間の考え方の違いは、実装における違いにつながります。

二項式、ブラック - ショールズ法、ヨーロピアン・モンテカルロ法では、初期データを生成する必要があります。oneAPI サンプルでは、インテル® oneMKL の乱数生成器 (英語) を用いて、各オプションの株価、権利行使価格、およびオプション年数の初期値を生成します。これはランダムな一様分布を用いて生成されます。この部分は通常計測されません。

```
...
  namespace mkl_rng = oneapi::mkl::rng;

  mkl_rng::philox4x32x10 engine(queue, rand_seed);

  sycl::event event_1 = mkl_rng::generate(
      mkl_rng::uniform<DATA_TYPE>(5.0, 50.0), engine, opt_n, h_stock_price);
  sycl::event event_2 = mkl_rng::generate(
      mkl_rng::uniform<DATA_TYPE>(10.0, 25.0), engine, opt_n, h_option_strike);
  sycl::event event_3 = mkl_rng::generate(
      mkl_rng::uniform<DATA_TYPE>(1.0, 5.0), engine, opt_n, h_option_years);
...
```

図 1. インテル® oneMKL を使用した初期乱数生成

二項式とブラック - ショールズ法の計算では、算術演算と、対数、平方根、指数などの標準的な数学関数のみを使用します。ベンチマークのカーネルには大規模な計算が含まれますが、各 SYCL ワークアイテムが 1 つのオプション（二項式）またはオプションの一部（ブラック - ショールズ）を処理するため、簡単に並列化できます。

```
queue->parallel_for(sycl::nd_range(...)),
  [=](sycl::nd_item<1> item) {
    for (.../* オプションの一部を処理 */) {
      const DATA_TYPE XexpRT = x * sycl::exp(-risk_free * t);
      DATA_TYPE n_d1 = DATA_TYPE(0.5) + DATA_TYPE(0.5) * sycl::erf(((sycl::log(s / x) +
(risk_free + DATA_TYPE(0.5) * sigma * sigma) * t) / (sigma * sycl::sqrt(t))) * sqrt1_2);
      DATA_TYPE n_d2 = DATA_TYPE(0.5) + DATA_TYPE(0.5) * sycl::erf(((sycl::log(s / x) +
(risk_free - DATA_TYPE(0.5) * sigma * sigma) * t) / (sigma * sycl::sqrt(t))) * sqrt1_2);
      const DATA_TYPE call_val = s * n_d1 - XexpRT * n_d2;
      const DATA_TYPE put_val = call_val + XexpRT - s;
      h_call_result_local[opt] = call_val;
      h_put_result_local[opt] = put_val;
    }
  }
);
```

図 2. ブラック - ショールズ・カーネルの擬似コード

ベンチマークの完全なコードは、ブラック - ショールズ・サンプル（英語）と二項式サンプル（英語）から入手できます。ヨーロピアン・モンテカルロ法では、株価、オプションの権利行使価格、オプション年数の初期値の生成に加えて、カーネル内部でランダムなガウス分布をその場で生成する必要があります。そのため、ベンチマークは二項式やブラック - ショールズよりも複雑になります。この生成には、以下の理由から、インテル® oneMKL 乱数生成器（RNG）デバイス API を使用します。

- 生成されたデータを格納するメモリーは必要ありません。ホスト API を使用した場合、約 90GB のメモリーが必要になる可能性があります。
- ガウス分布のパラメーターは、処理中のオプションに依存します。
- インテル® oneMKL のホスト API を使用すると、API 呼び出し内でカーネル送信によるオーバーヘッドが発生します。

```
namespace mkl_rng = oneapi::mkl::rng;
...
my_queue.parallel_for(
  /* sycl nd_range を設定*/,
  [=](sycl::nd_item<1> item)
  {
    /* オプション の一部を処理 */
    for(std::size_t i = 0; i < ITEMS_PER_WORK_ITEM; ++i)
    {
      ...
      mkl_rng::device::gaussian<DataType> distr(MuByT, VBySqrtT);

      /* 価格パスの一部を処理 */
      for (int block = 0; block < block_n; ++block)
      {
        auto rng_val_vec = mkl_rng::device::generate(distr, local_state);
        ...
      }
    }
  }
);
...
```

図 3. カーネル内での乱数生成

一部の乱数エンジン（例:MRG32k3a）では、コンストラクターの実行に時間がかかるため、あらかじめ別のカーネルでエンジンの状態を構築しておき、それらを生成処理とリダクションを含む後処理から構成される別のカーネル内で分布を得るために使うほうが効率的です。

```
namespace mkl_rng = oneapi::mkl::rng;
...

// 乱数エンジンの状態を初期化するカーネル
my_queue.parallel_for(
  sycl::range<1>(n_states),
  [=](sycl::item<1> idx) {
    auto id = idx[0];
    rng_states[id] = EngineType(/* エンジンのパラメーター */);
  }).wait_and_throw(); // 次のカーネルまでに処理を完了させる
```

```
...
// 計算された状態を使用して乱数を生成するカーネル
my_queue.parallel_for<k_MonteCarlo<DataType, ITEMS_PER_WORK_ITEM>>(
  /* sycl nd_range を設定 */,
  [=](sycl::nd_item<1> item)
  {
    auto local_state = rng_states[item.get_global_id()];

    for(std::size_t i = 0; i < ITEMS_PER_WORK_ITEM; ++i)
    {
      const std::size_t i_options = item.get_group_linear_id()*ITEMS_PER_WORK_ITEM+i;
      mkl_rng::device::gaussian<DataType> distr(MuByT, VBySqrtT);

      for (int block = 0; block < block_n; ++block)
      {
        auto rng_val_vec = mkl_rng::device::generate(distr, local_state);
        ...
      }
  }
);

...
```

図 4. 乱数エンジンの状態を個別に初期化

サンプルの最適化において、リダクションは興味深い部分です。生成された乱数の平均と標準偏差を計算する必要があります。ここでは、パフォーマンス上の理由から、各オプションの価格パスをワークグループの異なるワークアイテムに分割するため、SYCL のグループレデュースを呼び出す必要があります。もちろん、ほかのワークアイテムに依存しないように、すべての価格パスを 1 つのワークアイテムで処理するほうが簡単ですが、各オプションを 1 つのワークグループで処理し、価格パスをすべてのワークアイテムに均一に分散させるほうがパフォーマンスが向上することが判明したため、ここでは妥協案としてこのアプローチを採用しています。

```
namespace mkl_rng = oneapi::mkl::rng;
...
my_queue.parallel_for(
  /* sycl nd_range を設定 */,
  [=](sycl::nd_item<1> item)
  {
    /* オプションの一部を処理 */
    for(std::size_t i = 0; i < ITEMS_PER_WORK_ITEM; ++i)
    {
      ...
      mkl_rng::device::gaussian<DataType> distr(MuByT, VBySqrtT);

      /* 価格パスの一部を処理 */
      for (int block = 0; block < block_n; ++block)
      {
        auto rng_val_vec = mkl_rng::device::generate(distr, local_state);
        auto rng_val = Y * sycl::exp2(rng_val_vec) - Z;
        for (int lane = 0; lane < VEC_SIZE; ++lane)
        {
          DataType rng_element = sycl::max(rng_val[lane], DataType{});

          // ワークアイテム内でレデュース
          v0 += rng_element;
          v1 += rng_element * rng_element;
        }
      }

      // ワークグループ内でレデュースして値を計算
      v0 = sycl::reduce_over_group(group_id, v0, std::plus<>());
      v1 = sycl::reduce_over_group(group_id, v1, std::plus<>());
      ...
    }
  }
);
...
```

図 5. 各オプションの全価格パスのリダクション

ベンチマークの完全なコードは、ヨーロピアン・モンテカルロ・サンプル（英語）から入手できます。

アメリカン・モンテカルロは複数のカーネルを含むため、上記のサンプルよりも複雑です。インテル® oneMKL の RNG ホスト API のガウス分布と MRG32k3a 乱数エンジンを使用します。パフォーマンス向上のため、RNG デバイス API を使用することもできます。

ベンチマークの完全なコードは、アメリカン・モンテカルロ・サンプル（英語）から入手できます。

GitHub リポジトリー（英語）にある手順に従って実行することで、インテルのソフトウェアおよびハードウェアでサンプルが正しい結果を生成することを確認できます。

インテル® oneMKL で金融リスク評価を高速化

オープンソースの oneMath とその SYCL API、そしてインテル® oneMKL を活用することで、迅速な結果と信頼性の高い金融資産リスク予測を実現できます。

インテル® oneMKL を使用すると、さまざまなプラットフォーム・アーキテクチャー構成で共通の API と SYCL を使用できます。

この記事の情報は、金融アプリケーションだけでなく、クリティカル・パフォーマンス・パスで乱数を計算するほかのアプリケーションにも適用できます。

インテル® コンパイラーおよびインテル® oneMKL ソフトウェア開発チームは、日常的に課題への対応に取り組んでおり、インテルのソフトウェアを改善するため、より効率的なアルゴリズムと最適化手法を常に模索しています。

ツールの使用を開始するには、インテル® oneAPI マス・カーネル・ライブラリー（インテル® oneMKL）、oneAPI Math Library（oneMath）（英語）、および SYCLomatic（英語）を参照してください。

そして、インテル® oneAPI ベース・ツールキットとインテル® HPC ツールキットをダウンロードしてください。

インテル® oneMKL、oneAPI、SYCL の関連情報

- インテル® oneAPI マス・カーネル・ライブラリー（インテル® oneMKL）
- oneAPI Math Library（oneMath）（英語）
- インテル® oneMKL 乱数生成器デバイスルーチン（英語）

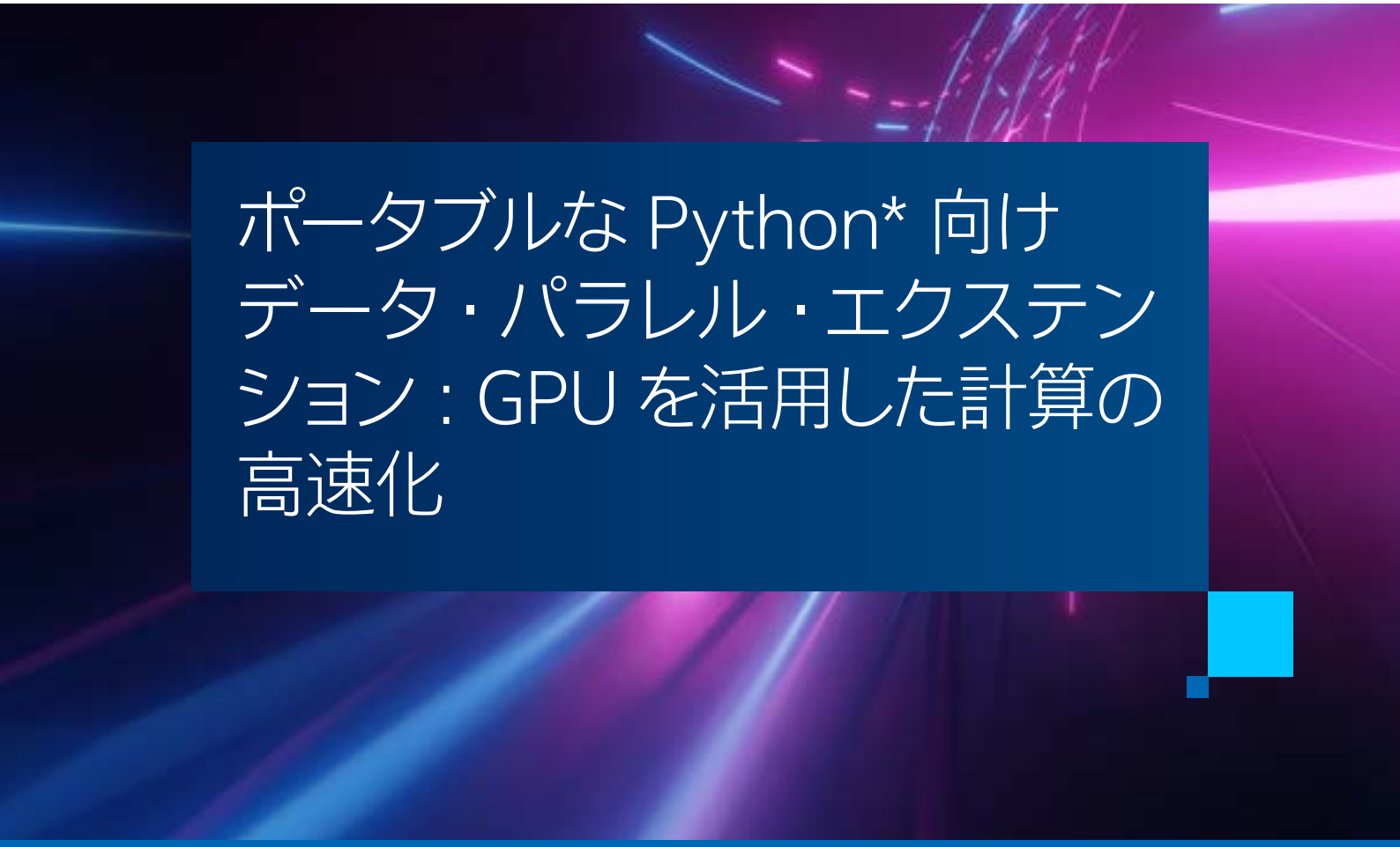
金融リスク計算の関連情報

- STAC-A2 : インテルの GPU を搭載した Dell PowerEdge サーバー上の oneAPI が金融市場のリスク分析を高速化（英語）
- 金融サービスのリスク計算における oneMKL 乱数生成器デバイス API
- Marco Dias : Real Options with Monte Carlo Simulation（英語）
- Rich Tanenbaum : Battle of the Pricing Models : Trees vs Monte Carlo（英語）
- Cox, J. C.; Ross, S. A. ; Rubinstein, M.（1979）. "Option pricing : A simplified approach（英語）". Journal of Financial Economics（英語）. **7**（3）: 229. DOI:10.1016/0304-405X（79）90015-1（英語）
- Rubinstein, M. (2000) . "On the Relation Between Binomial and Trinomial Option Pricing Models (英語)," Research Program in Finance, Working Paper Series（英語）, Research Program in Finance, Institute for Business and Economic Research, UC Berkeley.

インテル® Core™ Ultra プロセッサー搭載 AI PC で、より速くコーディング、よりスマートに創造力を発揮

生産性、創造性、コラボレーションを
強化する AI 搭載 PC

詳細



ポータブルな Python* 向けデータ・パラレル・エクステンション：GPU を活用した計算の高速化

Nikita Grigorian インテル コーポレーション AI フレームワーク・エンジニア
Oleksandr Pavlyk インテル コーポレーション AI フレームワーク・エンジニア

テクノロジーおよびソフトウェア業界でヘテロジニアス・プラットフォームが一般的になるにつれ、ベンダーやプラットフォームに依存しない方法でソフトウェアを開発し、パフォーマンス上のメリットを享受することが新たな課題となっています。この目的のため、Linux Foundation 傘下の Unified Acceleration Foundation（UXL）（英語）は、アクセラレーター・プログラミングのための、コンパイラーやパフォーマンス・ライブラリーを含むオープン・スタンダード・ソフトウェア・エコシステムを推進しています。この記事では、UXL ソフトウェア・エコシステムを Python に拡張し、ヘテロジニアス・プラットフォーム構成間での移植性とベンダー非依存を実現するとともに、ユーザーが同じ Python セッションで異なるベンダーのアクセラレーターを使って計算できるようにする方法について説明します。さらに、UXL Python エコシステムは、scikit-build や Meson などの標準 Python ツールと DPC++ コンパイラーを使用して、新しい移植可能なデータ並列ネイティブ Python 拡張機能を容易に作成できるようにします。

Python* 向けデータ・パラレル・エクステンションは、移植性の高い拡張機能の 1 つである `dpctl.tensor` を中心にして、統合共有メモリー（USM）割り当てに基づく配列オブジェクト `dpctl.tensor.usm_ndarray` と、配列オブジェクトを操作する関数ライブラリーを実装しています。移植性という目標に沿って、`dpctl.tensor` ライブラリーは、NumPy バージョン 2.0 以降、CuPy、Dask、その他のコミュニティー・プロジェクト（JAX、PyTorch、TensorFlow）などの NumFocus が支援するプロジェクト（英語）で積極的に採用されているテンソル・フレームワークの標準である Python Array API 標準（リビジョン 2023.12）（英語）に準拠するように設計されています。scikit-learn、SciPy など、従来は NumPy を利用して配列オブジェクトとそれを操作するライブラリーを提供していた Python パッケージは、Array API 準拠のライブラリーの配列オブジェクト・サポートを拡張し続けています。`dpctl.tensor.usm_ndarray` は、これらのパッケージで動作します。

データ・パラレル・コントロール Python パッケージ（dpctl）は、SYCL ベースの配列 API ライブラリーのリファレンス実装に加えて、Python からプラットフォーム列挙、デバイス選択、USM 割り当て、実行配置を容易にする DPC++ ランタイム・エンティティーへの Python バインディングも提供します。

dpctl は、Cython や pybind11 などの一般的な Python 拡張生成器との統合を提供し、データ並列ネイティブ拡張が Python オブジェクト型を操作し、それらを基礎となる C++ クラスにマップすることを可能にします。つまり、pybind11 では、`dpctl.SyclQueue` は `sycl::queue` に双方向にマップされ、`dpctl.tensor.usm_ndarray` は dpctl によって実装された `dpctl::tensor::usm_ndarray` C++ クラスにマップされます。

`dpctl.tensor` パッケージは純粋な SYCL を使用して実装されており、インテル® oneAPI DPC++ コンパイラーでビルドされています。このパッケージは、CodePlay の oneAPI for NVIDIA GPU（英語）と oneAPI for AMD GPU（英語）により、複数の SYCL ターゲットのビルドをサポートしており、ユーザーは同じ Python 環境で異なるベンダーのアクセラレーターをオフロードターゲットにすることができます。

インテルの GPU および CPU には SPIR-V オフロードセクション、NVIDIA GPU には NVPTX64 オフロードセクション、AMD GPU には AMDGCN が必要です。デフォルトでは、DPC++ は SPIR-V セクションのみを生成します。詳細は、SciPy 2024 でのプレゼンテーション（英語）を参照してください。プロジェクトが、対象のアクセラレーターに対応するドライバースタックに適したオフロードセクションを生成するようにコンパイルされている場合、このパッケージにより、Python ユーザーは DPC++ が対応するさまざまなデバイスと連携できます。

CUDA デバイス向けに dpctl をビルドするには、dpctl のドキュメント（英語）の手順に従ってください。

```
Python 3.12.4 | packaged by conda-forge | (main, Jun 17 2024, 10:23:07) [GCC 12.3.0]
Type 'copyright', 'credits' or 'license' for more information
IPython 8.26.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: import pandas as pd

In [2]: import dpctl

In [3]: pd.DataFrame([[d.name, d.filter_string] for d in dpctl.get_devices()])
Out[3]:
                                    0                 1
0  12th Gen Intel(R) Core(TM) i9-12900      opencl:cpu:0
1            Intel(R) UHD Graphics 770      opencl:gpu:0
2            Intel(R) UHD Graphics 770  level_zero:gpu:0
3               NVIDIA GeForce GT 1030        cuda:gpu:0
```

インテル® oneAPI DPC++ ランタイムが認識可能なデバイスは、dpctl でも認識されます。デフォルトでは、`dpctl.tensor` は SYCL のデフォルトセレクターによって選択されたデバイスをターゲットとします。特定のデバイスを選択するには、インテル® oneAPI SYCL 拡張のフィルターセレクター（英語）を使用します。フィルターセレクター文字列「`backend:device_type:ordinal_id`」は 3 つの要素で構成されており、少なくとも 1 つの要素が指定されていれば、残りの要素は省略できます。以下に、異なるデバイス上に算術シーケンスの値を格納する配列を作成する例を示します。

```
In [1]: import dpctl.tensor as dpt

In [2]: x_cuda = dpt.arange(6, device="cuda"); (x_cuda, x_cuda.device)
Out[2]: (usm_ndarray([0, 1, 2, 3, 4, 5]), Device(cuda:gpu:0))

In [3]: x_cpu = dpt.arange(6, device="cpu"); (x_cpu, x_cpu.device)
Out[3]: (usm_ndarray([0, 1, 2, 3, 4, 5]), Device(opencl:cpu:0))

In [4]: x_gpu = dpt.arange(6, device="gpu"); (x_gpu, x_gpu.device)
Out[4]: (usm_ndarray([0, 1, 2, 3, 4, 5]), Device(level_zero:gpu:0))

In [5]: x_default_device = dpt.arange(6); (x_default_device, x_default_device.device)
Out[5]: (usm_ndarray([0, 1, 2, 3, 4, 5]), Device(level_zero:gpu:0))
```

配列の作成時に、デバイス・キュー・キャッシュから SYCL キューが暗黙的に割り当てられます。このキューは、項目の割り当て（`(x[:] = 0)`）など、配列要素を操作するタスクの実行に使用されます。ユーザーは、すべての配列作成関数でサポートされている `sycl_queue` キーワード引数を介して、このようなキューを明示的に指定することもできます。

`compute-follows-data` は、`dpctl.tensor` で採用されているパラダイムであり、ユーザーは配列作成関数でデバイス配置を指定するだけで利用できます。ほかのすべての関数の出力は同じデバイスに割り当てられ、入力配列と同じキューに関連付けられます。また、すべての入力配列は同じキューに関連付けられることが期待されます。

```
In [17]: x_cuda = dpt.linspace(0, 1, num=6, device="cuda")

In [18]: y_cuda = dpt.sin(x_cuda)

In [19]: x_cpu = dpt.linspace(0, 1, num=6, device="cpu")

In [20]: y_cpu = dpt.sin(x_cpu)

In [21]: y_cuda.device, y_cpu.device
Out[21]: (Device(cuda:gpu:0), Device(opencl:cpu:0))

In [22]: y_cuda
Out[22]:
usm_ndarray([0.        , 0.19866933, 0.38941834, 0.56464247, 0.71735609,
             0.84147098])

In [23]: y_cpu
Out[23]:
usm_ndarray([0.        , 0.19866933, 0.38941834, 0.56464247, 0.71735609,
             0.84147098])
```

異なるデバイスに存在する配列を結合する必要がある場合は、`usm_ndarray` の `to_device(target_dev)` メソッドまたは `dpctl.tensor.asarray` コンストラクターを使用して、明示的にデータ移行を実行する必要があります。

```
ExecutionPlacementError: Execution placement can not be unambiguously inferred from input arguments.

In [25]: y_cpu * y_cuda.to_device(y_cpu.device)
Out[25]:
usm_ndarray([0.        , 0.0394695 , 0.15166665, 0.31882112, 0.51459976,
             0.70807342])

In [26]: _.device
Out[26]: Device(opencl:cpu:0)

In [27]: dpt.asarray(y_cpu, device=y_cuda.device) * y_cuda
Out[27]:
usm_ndarray([0.        , 0.0394695 , 0.15166665, 0.31882112, 0.51459976,
             0.70807342])

In [28]: _.device
Out[28]: Device(cuda:gpu:0)
```

`(y_cpu * y_cuda)` を計算しようとすると、`ExecutionPlacementError` が発生します。明示的にデータを移行してこの問題を解決します。

```
In [32]: y = dpt.exp(-x*x/2)*dpt.sin(dpt.pi * x) + dpt.exp(-x*x/3)*dpt.cos(dpt.pi * x/3)

In [33]: x[dpt.argmax(y)]
Out[33]: usm_ndarray(0.39201736, dtype=float32)

In [34]: x[dpt.argmin(y)]
Out[34]: usm_ndarray(-0.5377215, dtype=float32)
```

この曖昧さは、入力配列の作成時にデフォルトでないデバイスに配置された場合にのみ発生します。デフォルトでは、すべての配列はデフォルトで選択された同じデバイス上に作成され、同じキューに関連付けられます。

dpctl.tensor は Python Array API 仕様に準拠しているため、Array API をサポートするコミュニティー・パッケージは dpctl.tensor.usm_ndarray オブジェクトを処理できます。例として、Array API の試験的なサポートを含む SciPy の FFT パッケージを使用して、usm_ndarray の高速フーリエ変換を計算してみましょう。

```
(dev_dpctl) opavlyk@orsatdevnuc01:/localdisk/work/opavlyk/repos/dpctl$ SCIPY_ARRAY_API=1 ipython
Python 3.12.4 | packaged by conda-forge | (main, Jun 17 2024, 10:23:07) [GCC 12.3.0]
Type 'copyright', 'credits' or 'license' for more information
IPython 8.26.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: import dpctl.tensor as dpt

In [2]: import scipy.fft

In [3]: x = dpt.linspace(-dpt.pi/2, dpt.pi/2, num=2**16)

In [4]: s = dpt.sin(2*x)

In [5]: omega = scipy.fft.fft(s)

In [6]: omega.device
Out[6]: Device(level_zero:gpu:0)

In [7]: omega[ dpt.abs(omega) < dpt.max(dpt.abs(omega))/1000 ] = 0

In [8]: omega[:16]
Out[8]:
usm_ndarray([ 0.          +0.j  , -1.5707895+32767.75j],
              0.          +0.j  ,  0.          +0.j  ,
              0.          +0.j  ,  0.          +0.j  ,
              0.          +0.j  ,  0.          +0.j  ,
              0.          +0.j  ,  0.          +0.j  ,
              0.          +0.j  ,  0.          +0.j  ,
              0.          +0.j  ,  0.          +0.j  ], dtype=complex64)
```

dpctl.tensor は、統合 GPU とディスクリート GPU の両方で計算の高速化に役立ちます。以下は、Windows 上のインテル® Arc™ GPU と Linux 上のインテル® データセンター GPU Max 1100（ディスクリート GPU）で実行された K 近傍法（KNN）検索ベンチマーク・コード（英語）の例です。conda-forge からインストールされた標準の NumPy と比較して、それぞれ 2.5 倍と 31 倍の高速化が示されています。

```
Problem description:
    n_obs  = 1211223
    n_poi  = 1125
    dim    = 11
    k      = 3
Result agreement:  True
Execution wall times:
  NumPy         : 111.1419189000735 seconds
     version: 2.1.1
  dpctl.tensor : 44.219180100015365 seconds,
     version: 0.19.0dev0+10.ged7d41e11a
     device : Intel(R) Arc(TM) 140V GPU (16GB)
```

```
Problem description:
    n_obs  = 1211223
    n_poi  = 1125
    dim    = 11
    k      = 3
Result agreement:  True
Execution wall times:
  NumPy         : 116.42102308997911 seconds
     version: 2.0.1
  dpctl.tensor : 3.7495630119228736 seconds,
     version: 0.19.0dev0+34.g109b4c289a
     device : Intel(R) Data Center GPU Max 1100
```

PCA と DBSCAN による時系列クラスタリング

scikit-learn* 向けインテル® エクステンションを利用したアルゴリズムの高速化

Bob Chesebrough インテル コーポレーション シニア AI ソリューション・アーキテクト

この記事では、次元縮小に主成分分析（PCA）、クラスタリングにノイズを含むアプリケーションの密度ベースの空間クラスタリング（DBSCAN）を用いて、時系列データをクラスタリングする方法について考察します。この手法は、ラベル付きデータなしで、都市の交通量などの時系列データ内のパターンを識別します。パフォーマンス向上のため、scikit-learn* 向けインテル® エクステンション（英語）を使用します。

時系列データには、人間の行動、機械、その他の測定可能なソースによる反復パターンがしばしば見られます。このようなパターンを手作業で識別することは困難です。PCA や DBSCAN などの教師なし学習アプローチは、これらのパターンの発見を可能にします。

手法

データ生成

時系列パターンをシミュレーションするため、合成波形データを生成します。データは 3 つの異なる波形で構成され、それぞれにノイズを追加することで、実世界の変動をシミュレーションします。Gaël Varoquaux 氏が作成した scikit-learn の凝集型クラスタリングの例を使用します（**図 1**）。これは、BSD-3-Clause（英語）または CC0（英語）ライセンスの下で利用可能です。

```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(0)
n_features = 2000
t = np.pi * np.linspace(0, 1, n_features)

def sqr(x):
    return np.sign(np.cos(x))

X = []
y = []
for i, (phi, a) in enumerate([(0.5, 0.15), (0.5, 0.6), (0.3, 0.2)]):
    for _ in range(30):
        phase_noise = 0.01 * np.random.normal()
        amplitude_noise = 0.04 * np.random.normal()
        additional_noise = 1 - 2 * np.random.rand(n_features)
        additional_noise[np.abs(additional_noise) < 0.997] = 0
        X.append(12 * ((a + amplitude_noise)

                       * (sqr(6 * (t + phi + phase_noise)))

                       + additional_noise))
        y.append(i)

X = np.array(X)
y = np.array(y)

plt.figure()
plt.axes([0, 0, 1, 1])
for l in range(3):
    plt.plot(X[y == l].T, alpha=0.5, label=f'Waveform {l+1}')
plt.legend(loc='best')
plt.title('Unlabeled Data')
plt.show()
```

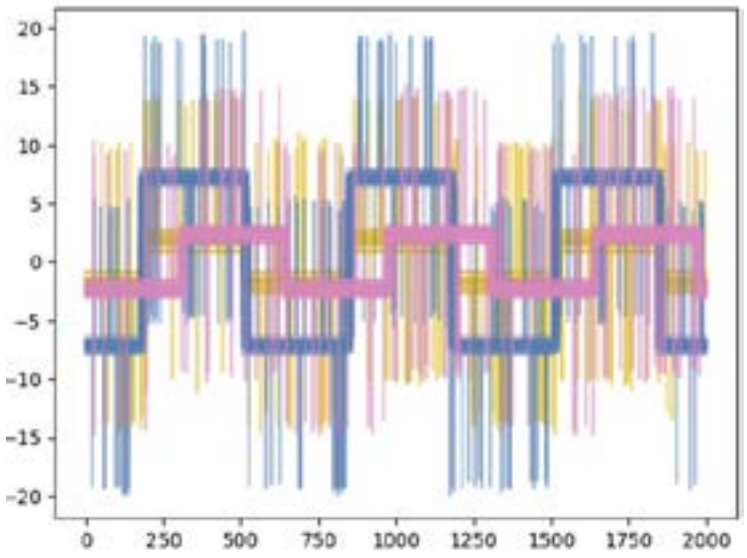


図 1. Gaël Varoquaux 氏が作成した scikit-learn 凝集型クラスタリング・アルゴリズムから著者が生成したコードとプロット

scikit-learn* 向けインテル® エクステンションを利用したアルゴリズムの高速化

PCA と DBSCAN はどちらも、scikit-learn* 向けインテル® エクステンションを用いたパッチ適用スキームによって高速化できます。scikit-learn（sklearn とも呼ばれる）は、マシンラーニング（ML）用の Python モジュールです。scikit-learn* 向けインテル® エクステンションは、シングルノード構成およびマルチノード構成のインテルの CPU および GPU 上で scikit-learn アプリケーションをシームレスに高速化するインテルの AI ツール（英語）の 1 つです。scikit-learn 推定器に動的にパッチを適用することで、ML のトレーニングと推論を同等の数学的精度で最大 100 倍向上させます（**図 2**）。

図 2. scikit-learn* 向けインテル® エクステンションの GitHub リポジトリー（英語）

scikit-learn* 向けインテル® エクステンションは scikit-learn API を使用し、scikit-learn をインポートする前に Python アプリケーションを数行変更することで有効にできます。

```
from sklearnex import patch_sklearn
patch_sklearn()
```

PCA を用いた次元縮小

2,000 個の特徴を含む 90 個のサンプルをクラスタリングする前に、データセットの分散の 99% を維持しながら PCA を用いて次元を縮小します。

```
from sklearn.decomposition import PCA

pca = PCA(n_components=4)
XPC = pca.fit_transform(X)

print("Explained variance ratio:", pca.explained_variance_ratio_)
print("Singular values:", pca.singular_values_)
print("Shape of XPC:", XPC.shape)
```

ペアプロットを使用して、縮小されたデータからクラスターを探します（**図 3**）。

```
import pandas as pd
import seaborn as sns

df = pd.DataFrame(XPC, columns=['PC1', 'PC2', 'PC3', 'PC4'])
sns.pairplot(df)
plt.show()
```

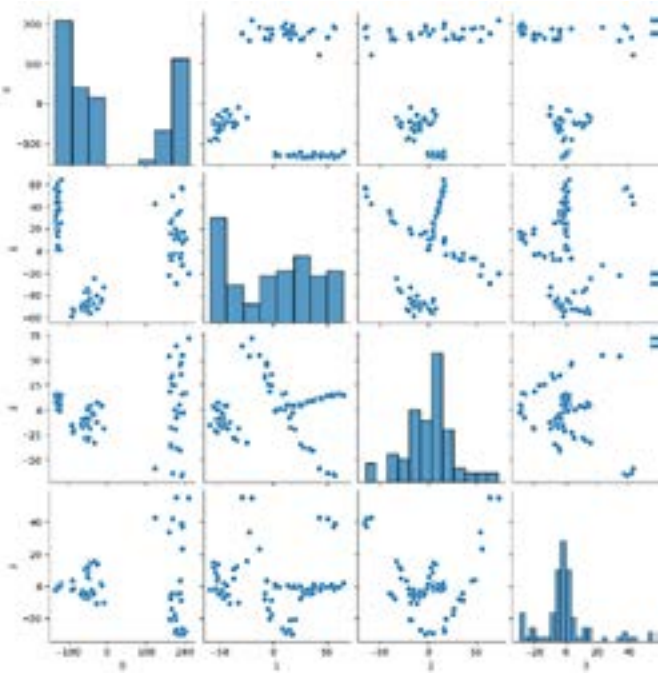


図 3. 次元縮小後のデータからクラスターを見つける

DBSCAN を用いたクラスタリング

ペアプロットから、PC1 と PC2 はクラスターを適切に分離しているため、これらの要素を DBSCAN クラスタリングに使用します。DBSCAN EPS パラメーターの値も推定できます。PC1 と PC0 の図から、観察されたクラスターでは 50 が妥当な分離距離であることが分かります。

```
from sklearn.cluster import DBSCAN

clustering = DBSCAN(eps=50, min_samples=3).fit(XPC[:, [0, 1]])
labels = clustering.labels_

print("Cluster labels:", labels)
```

クラスター化されたデータをプロットすると、DBSCAN がクラスターをどの程度正確に識別したかを確認できます (**図 4**)。

```
plt.figure()
plt.axes([0, 0, 1, 1])
colors = [ "#f7bd01" , "#377eb8" , "#f781bf" ]
for l, color in zip(range(3), colors):
    plt.plot(X[labels == l].T, c=color, alpha=0.5, label=f' Cluster {l+1}' )
plt.legend(loc=' best' )
plt.title( 'PCA + DBSCAN' )
plt.show()
```

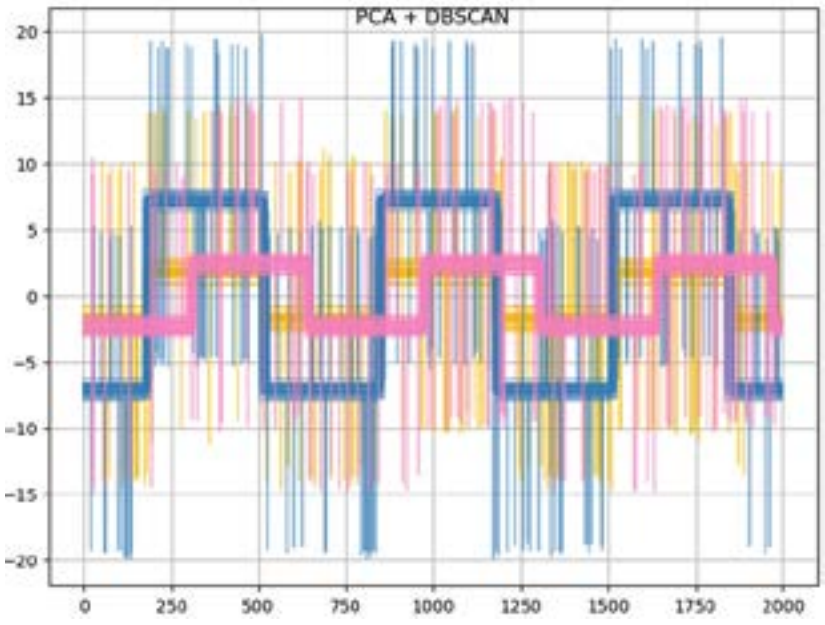


図 4. 前のコード例を使用して生成されたクラスター化データのプロット

元のデータとの比較

図 4 から分かるように、DBSCAN は妥当な色のクラスターを検出し、元のデータ（**図 1**）と一致しています。このケースでは、クラスタリングによってデータ生成に使用された基本パターンが完全に復元されました。次元縮小に PCA を使用し、クラスタリングに DBSCAN を使用することで、時系列データ内のパターンを効果的に識別し、ラベル付けすることができます。このアプローチにより、ラベル付きサンプルがなくても、データの基本構造を発見することができます。

Transformer ベースの時系列予測器の実装

インテル® Tiber™ AI クラウドによる時系列分析の簡素化

Bob Chesebrough インテル コーポレーション シニア AI ソリューション・アーキテクト
Jack Erickson インテル コーポレーション 主席デベロッパー・マーケティング・マネージャー

最近、時系列分析に関心のある複数のクライアントと仕事をする機会がありました。これは、時系列予測とクラスタリングの最新および従来の手法を探求する絶好の機会となりました。そこで得た知識を、時系列予測に関するこの記事と、本号の記事「**PCA と DBSCAN による時系列クラスタリング**」に分けて紹介します。

Hugging Face コミュニティーには、時系列予測用の Chronos（英語）モデルがあります。Chronos は、時系列予測モデルのファミリーで、言語モデルが次のトークンの予測に使用している Transformer アーキテクチャーに基づいています。このモデルは、公開されている大規模な時系列データコーパスで事前トレーニング済みであり、データに合わせてファイン・チューニングが可能です。ここでは、AutoGluon（英語）ライブラリーの chronos-bolt-small（英語）モデルを使用します。以下は、AutoGluon のウェブページから抜粋した機能に関する簡単な説明です。

「シンプルな `fit()` を呼び出すだけで、AutoGluon はシンプルな予測モデル（ARIMA、ETS、Theta など）、強力なディープラーニング・モデル（DeepAR、Temporal Fusion Transformer など）、ツリーベース・モデル（LightGBM など）、およびその他のモデルの予測を組み合わせて、一変量時系列データの複数ステップ先の確率予測を生成するアンサンブルをトレーニングおよびチューニングできます。」

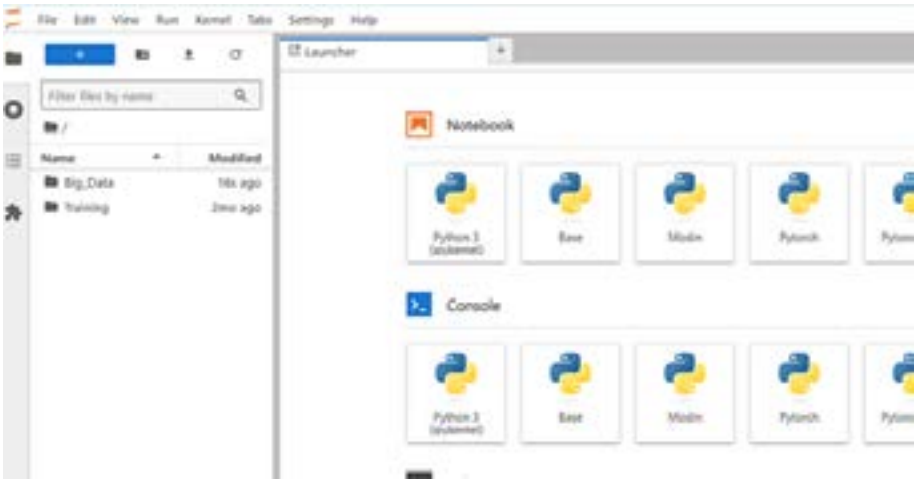
インテル® Tiber™ AI クラウド

この例では、最新のインテルの AI ハードウェアとソフトウェア（英語）を利用できるインテル® Tiber™ AI クラウドの無料アカウントを使用します。初期プロジェクトの実行手順を以下に示します。この例では、インテル® Xeon® スケーラブル・プロセッサーを使用します。インテル® Tiber™ AI クラウドのアカウントをお持ちでない場合は、無料で登録できます。

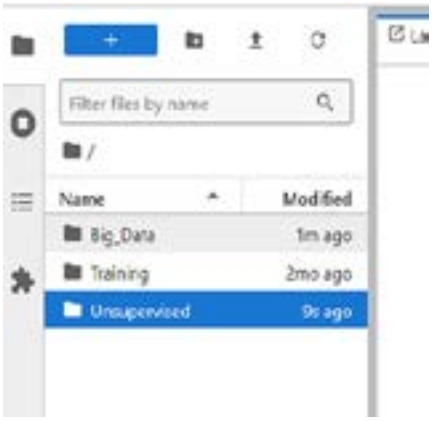
初期プロジェクトのセットアップ

アカウントを作成したら、以下の手順に従ってください。

1. 「Get Started」（英語）にアクセスします。この項目は無料の学習アカウントで開始できます。
2. **[Connect Now]** をクリックし、任意のデバイスタイプ（この例では CPU のみを使用）を選択して JupyterLab を起動します。
3. 以下のような JupyterLab インターフェイスが表示されます。

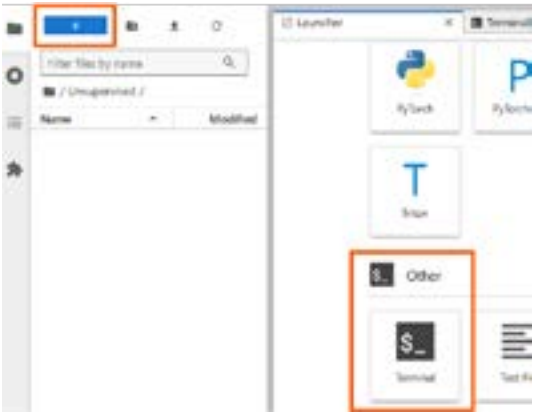


4. 以下のように「`Unsupervised`」という名前のフォルダーを作成します。

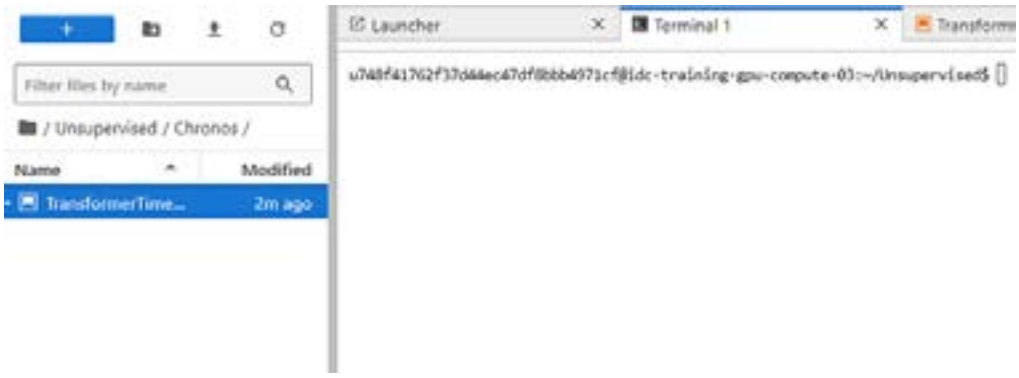


次のステップに進む前に「`Unsupervised`」をダブルクリックします。

5. プロジェクトのコードは、Git リポジトリー ChronosTimeSeriesPredictor（英語）にあります。Git コマンドを実行してリポジトリーのクローンを作成するため、ターミナルセッションを起動します。



これにより、Bash シェルターミナルのブラウザータブが作成されます。



次のコマンドで、ディレクトリー内にリポジトリーをクローンします。

```
git clone https://github.com/jackerickson1/ChronosTimeSeriesPredictor.git
```

仮想環境と Jupyter カーネルの作成

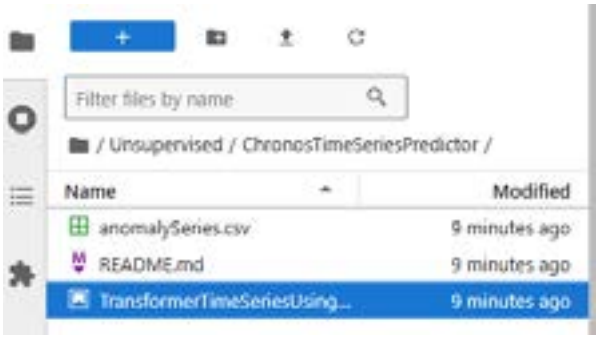
ターミナルで以下のコマンドを実行して、専用の Python 仮想環境を作成し、必要なライブラリーをインストールし、仮想環境に関連付けられた Jupyter カーネルを作成します。

```
python -m venv venv_timeseries
source venv_timeseries/bin/activate
pip install -q torch torchvision --index-url https://download.pytorch.org/whl/cpu
pip install -q autogluon ipywidgets ipykernel
python -m ipykernel install --user --name timeseries
```

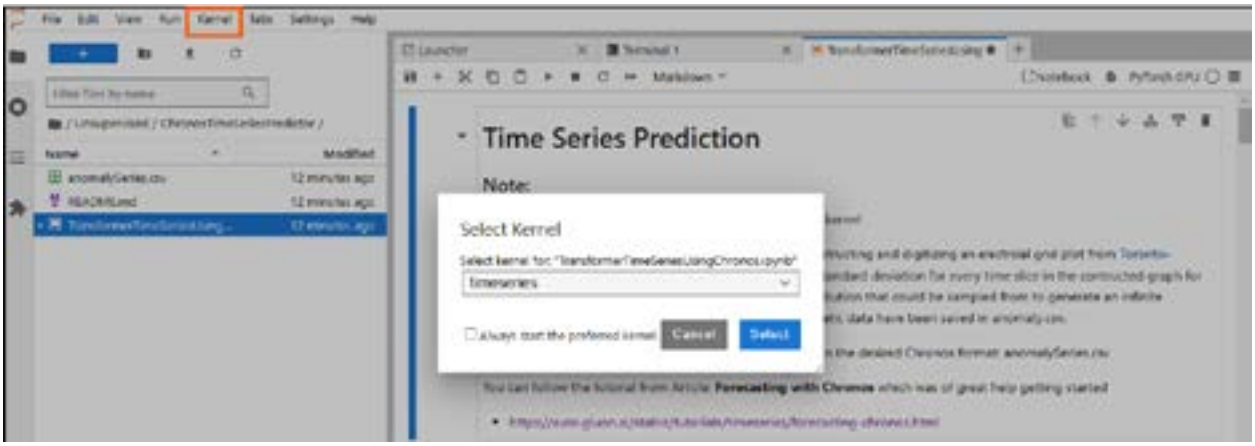
これらのパッケージのインストールには数分かかります。ノートブックを開いたら、「timeseries」カーネルを使用してコードを実行します。

分析用のノートブックを開く

`ChronosTimeSeriesPredictor` フォルダーをダブルクリックして、その中の `TransformerTimeSeriesUsingChronos.ipynb` ノートブックを開きます。



ノートブックが開いたら、**[Kernel] > [Change Kernel…]** ドロップダウン・メニューから「timeseries」カーネルを選択し、新しく作成したカーネルに変更します。



ライブラリーのインポート

最初のコードセルを実行して、インストールされた Python ライブラリーから必要なモジュールをインポートします。

```
from autogluon.timeseries import TimeSeriesDataFrame, TimeSeriesPredictor
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import torch
from tqdm.auto import tqdm
```

データの取得と調査

このサンプルでは、Toronto-Hydro-Transformer-Monitors（英語）にある Toronto Hydro の電力網データを基に別のプロジェクトで作成された時系列データを使用しています。

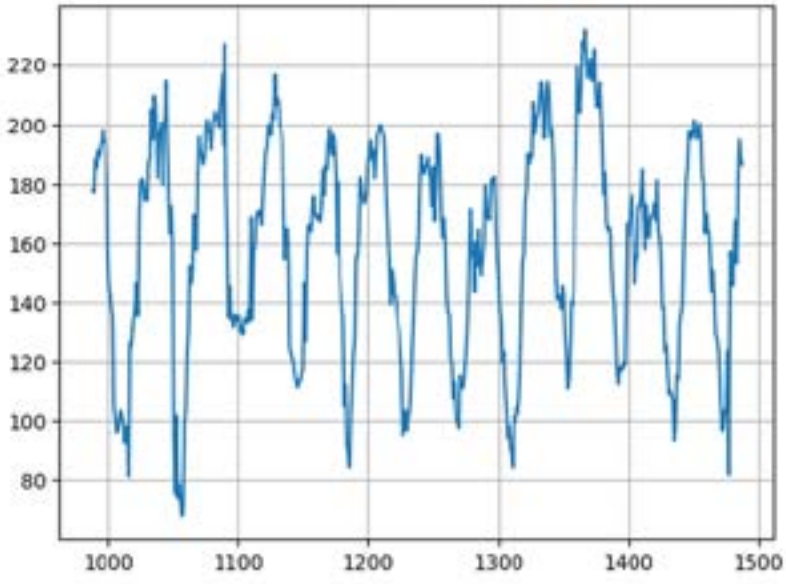
毎日の規則的な周期性と、統計的な範囲内で制約され変動する電力網の値の不規則な性質に注目してください。このデータは、複数日にわたり、毎日 10 分間隔で平均値と標準偏差を追跡して生成されました。各時間スライスはそれぞれ独自の平均値と標準偏差を持つものとしてモデル化され、ガウス分布が仮定されました。その後、この分布をサンプリングして無限量のデータが生成されました。

```
import pandas as pd
df = pd.read_csv(“anomalySeries.csv”)
plt.plot( df.target[-500:])
plt.grid()
plt.show()
```

時系列形式の CSV ファイルを読み込みます。

```
# TimeSeries 形式に変換
data = TimeSeriesDataFrame(“anomalySeries.csv”
```

ゼロショット予測器の作成



モデルをそのままデータに適用するため、予測をゼロショットモードで実行するように設定します。モデルはすでに大量の時系列データでトレーニング済みであり、ファイン・チューニングによってカスタマイズを行うのに十分なトレーニング・データがありません。代わりに、トレーニング用データのサブセットを予測器への入力として使用します。予測器の予測期間は 24 トークンです。

AutoGluon にはモデル選択機能がありますが、ここでは「bolt_small」モデルを使用するように指示します。

```
prediction_length = 24
train_data, test_data = data.train_test_split(prediction_length)

predictor = TimeSeriesPredictor(prediction_length=prediction_length).fit(
    train_data, presets=”bolt_small”,
)
```

テストデータの予測

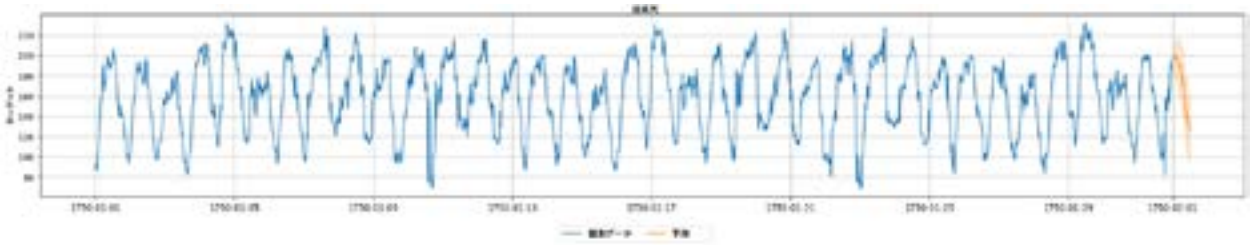
最後に、テストのサブセットを使って、予測器が未知のデータに対してどの程度一般化できるかを測定します。

```
predictions = predictor.predict(test_data)

# matplotlib インタラクティブ・モードをオフにする
plt.ioff()

Len = data.shape[0]
predictor.plot(
    data=data,
    predictions=predictions,
    item_ids=[“Series”],
    max_history_length=Len,
)
```

プロット出力は、トレーニングされたモデルが周期性を適切に予測していることを示しています。



まとめ

インテル® Tiber™ AI クラウドの無料アカウントを取得し、Chronos モデルを使用して時系列を簡単に素早くトレーニングおよび予測することができます。ぜひお試しください。

JAX と OpenXLA の実行プロセスと基本ロジック

パート 1

Zhenming Wang、Lu Teng、Yabai Hu、Yi Zhou、Lifeng Wang、Yi Yao、Jack Chen、Xuming Gai、Wenjun Liu インテル コーポレーション AI フレームワーク チーム

この記事では、JAX フレームワークと OpenXLA バックエンドにおける Python プログラムの実行ワークフローと、基本となる動作ロジックについて考察します。まず、OpenXLA の高レベル統合構造と JAX の基本的な概念の概要を説明します。インテルの GPU 上で OpenXLA を使用した JAX の実行例を示すことで、Python プログラムが JAX フレームワークによってどのように認識され、StableHLO 表現に変換されるか詳細な分析を提供します。次に、OpenXLA はこれらの StableHLO 表現を HLO（高水準操作）に解析します。

この記事では、JAX フレームワークと OpenXLA コンパイラーがどのように Python プログラムを解釈するかについて初期考察を提供し、より複雑なモデルの実行と分析の基礎を築きます。これは、インテルの GPU 上で OpenXLA コンパイラー開発に取り組む開発者や、JAX/OpenXLA モデルのデバッグに携わる開発者にとって、基礎的なガイドとなるでしょう。

JAX : Auto-grad と XLA

JAX（英語）は、高性能数値計算と大規模マシンラーニング向けに設計された、アクセラレーター指向の配列計算とプログラム変換用の Python ライブラリーです。XLA（Accelerated Linear Algebra）（英語）は、マシンラーニング向けのオープンソース・コンパイラーです。

OpenXLA* 向けインテル® エクステンション

OpenXLA（英語）プロジェクトは、開発者コミュニティーと主要な人工知能（AI）およびマシンラーニング（ML）チームを結集して、ML を加速し、ML フレームワークとハードウェア間のインフラストラクチャーの断片化に対処します。

OpenXLA* 向けインテル® エクステンション（英語）には、インテルの GPU 上で JAX モデルをシームレスに実行する PJRT プラグインが実装されています。PJRT API は統合を簡素化するため、インテルの GPU 用のプラグインを個別に開発し、JAX に迅速に統合することができます。この PJRT 実装により、XLA アクセラレーションを使用した TensorFlow および PyTorch モデルにおけるインテルの GPU の初期サポートも可能になります。詳細は、「RFC : OpenXLA PJRT プラグイン」（英語）を参照してください。

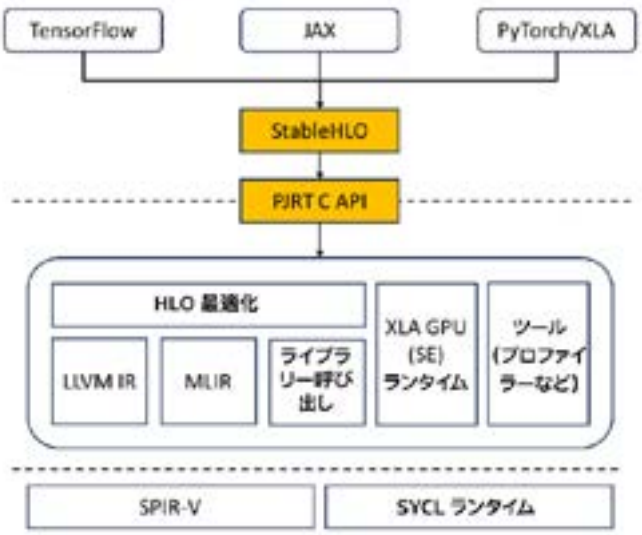


図 1. PJRT プラグインが実装された OpenXLA* 向けインテル® エクステンション（出典（英語））

JAX と OpenXLA* 向けインテル® エクステンションの組み合わせにより、インテルの GPU においてハードウェア適応とモデル・アクセラレーションが向上します。

実際の実行ロジック

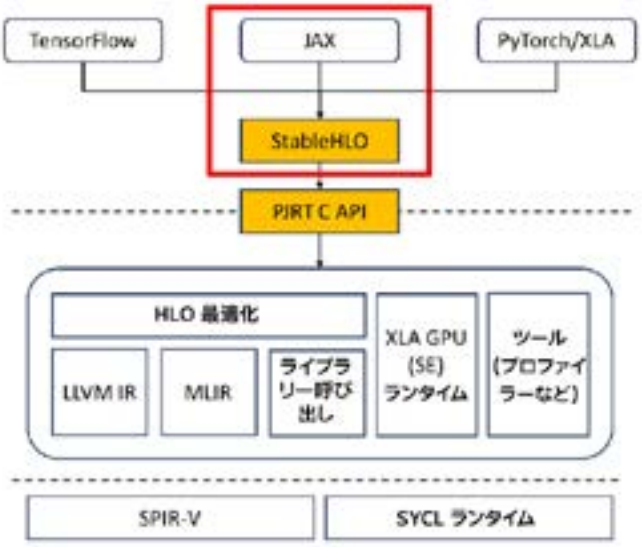
JAX フレームワークが Python プログラムを解析し、OpenXLA が計算グラフを生成し、演算は GPU ハードウェア上で実行されます。

以下の実行プロセスと詳細は、JAX と OpenXLA の開発およびデバッグのプロセスを示しています。プロセス全体は、JAX 表現、StableHLO、初期 HLO、最終 HLO、初期 LLVM IR、最終 LLVM IR、および SPIR-V ファイルの各フェーズに分かれています。

それでは、実際のコード例を使って説明しましょう。

```
import jax.numpy as jnp
from jax import grad
def simple_function(x):
    return x**2 + 3*x + 2
gradient = grad(simple_function)
x = 2.0
computed_gradient = gradient(x)
print("Computed Gradient:", computed_gradient)
```

JAX 表現のダンプから StableHLO ファイルへの変換



インテルの OpenXLA は現在、公開されている StableHLO モジュールを直接呼び出しており、追加の変更や最適化は行っていません。公開されている StableHLO モジュールは、JAX 表現の変換を行います（詳細は、このモジュールの説明（英語）を参照してください）。JAX フレームワークの開発では、StableHLO 表現に力を入れています。

JAX フレームワークはまず、Python 表現を StableHLO 表現に変換します。`export JAX_DUMP_IR_TO="dump"` を設定すると、`jax_ir0_xxx.mlir` のようなテキストファイルが生成され、式の実行プロセスの詳細が記録されます。詳細は、OpenXLA 公式ドキュメント（英語）を参照してください。

```
$ export JAX_DUMP_IR_TO="dump" python grad.py
```

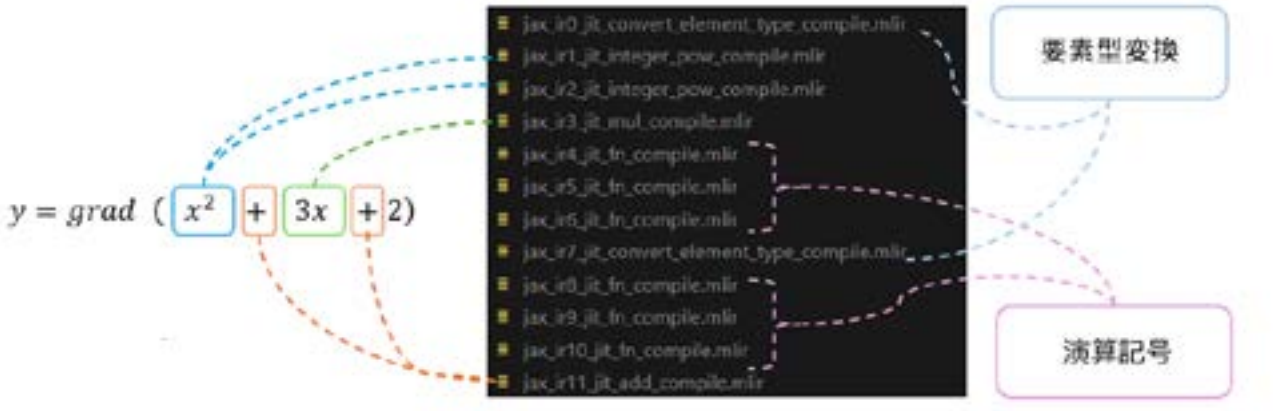


図 2. JAX 表現の変換例

OpenXLA は主に StableHLO 以降のフェーズを行うため、以降のステップについて簡単に説明します。

- `jax_ir0_jit_convert_element_type_compile.mlir`

  シンプルな JIT（ジャストインタイム）コンパイルモジュール `@jit_convert_element_type` が定義されています。このモジュールには `@main` 関数が含まれています。この関数は浮動小数点テンソルを入力として受け取り、何も処理せずにテンソルを直接返します。

  モジュール `@jit_convert_element_type` には、`mhlo.num_partitions` や `mhlo.num_replicas` などの属性があり、どちらも 1 に設定されていて、型は i32（32 ビット整数）です。これらの属性は、JIT コンパイル時に並列計算を行うパーティションとレプリカの数を指定します。

  このコード行は、MLIR（Multi-Level Intermediate Representation）言語を使用したモジュール宣言です。MLIR は、コンパイラーで計算グラフの構築と最適化に使用される、マルチレベル中間表現向けに設計されたフレームワークです。JAX やその他のディープラーニング・フレームワークは、MLIR とそのサブセット（MHLO や StableHLO など）を活用して効率的な計算を実現します。

- `jax_ir1_jit_integer_pow_compile.mlir`

  `X^2` の処理：このステップには、MLIR を用いて表現される単純なテンソル演算が含まれます。ここでは JIT コンパイラーを使って実行時に入力テンソルの 2 乗を計算します。`tensor<f32>` は 32 ビット浮動小数点テンソルを表します。

  `@jit_integer_pow` モジュールにはいくつかの属性があり、`mhlo.num_partitions` と `mhlo.num_replicas` はどちらも 1 に設定されています。これは、整数の累乗（例：`x^y`）を計算する関数または演算です。ここで、`x` は底、`y` は指数です。この演算は、数値を指定された整数の累乗にする必要がある数学的または科学的な計算でよく使用されます。

テンソル乗算:`stablehlo.multiply`（乗算を表す安定した高水準演算）を使用して、`%arg0` を自身で乗算し、その結果を `%0` に格納します。`loc(#loc7)` は、`#loc7` としてマークされたこの演算の位置を示します。

関数：この演算は実際のテンソル乗算を実行します。つまり、入力テンソル `%arg0` の 2 乗を計算します。位置マーカー `#loc7` は、この乗算のソースコード・コンテキストを追跡するのに役立ちます。

`#loc(numbers)`：位置マーカーは、コードの追跡だけでなく、最適化やデバッグ時にさまざまなステップや演算の発生源を理解するのにも役立ちます。

- `jax_ir2_jit_integer_pow_compile.mlir`
  `jax_ir3_jit_mul_compile.mlir`
  `jax_ir11_jit_add_compile.mlir`

  定義のこれらの部分は、上記のファイルに似ています。
- `jax_ir4_jit_fn_compile.mlir`
  `jax_ir5_jit_fn_compile.mlir`
  `jax_ir6_jit_fn_compile.mlir`

  簡略化された計算グラフは、MLIR と StableHLO を用いて表現されます。`@jit_fn` モジュールと、`@main` 関数内の乗算、加算、データ型変換などの計算プロセスを定義します。対応する演算マーカーは、デバッグや最適化時に追跡と位置特定に使用されます。
- `jax_ir7_jit_convert_element_type_compile.mlir`

  MLIR 構文を用いた JIT コンパイルプロセスは、データ型変換操作を行います。`stablehlo.convert` 命令を使用して、入力テンソル `%arg0` を同じ型（`tensor<f32>`）のテンソルに変換します。入力型と出力型は同じ（`tensor<f32>`）であるため、この変換によってデータ表現は変更されません。

以下にリストするファイルは、テストや簡単なデータフロー検証時にデータ転送および出力構造として使用されます。

`jax_ir8_jit_fn_compile.mlir`
`jax_ir9_jit_fn_compile.mlir`
`jax_ir10_jit_fn_compile.mlir`

ファイルと操作

共通機能

- **モジュールと関数の定義：**各コードブロックは、`@jit_fn` モジュールを定義します。このモジュールには `@main` 関数が含まれます。この関数は public としてマークされているため、モジュールの外部から呼び出すことができます。
- **JIT コンパイル：**`jit` は JAX の修飾子であり、JIT コンパイルを通じて関数を最適化するのに使用します。これにより、Python インタープリターのオーバーヘッドが軽減され、ハードウェア（CPU や GPU など）の計算効率が向上します。
- **テンソル：**テンソル（`tensor`）は関数が扱う主要なオブジェクトであり、各コードスニペットでは 32 ビット浮動小数点テンソル（`tensor<f32>`）として扱われます。
- **位置情報：**各コードスニペットには位置情報（`#loc` など）が含まれており、主にデバッグや最適化時にコードのソース位置の追跡に使用されます。

実行プロセス

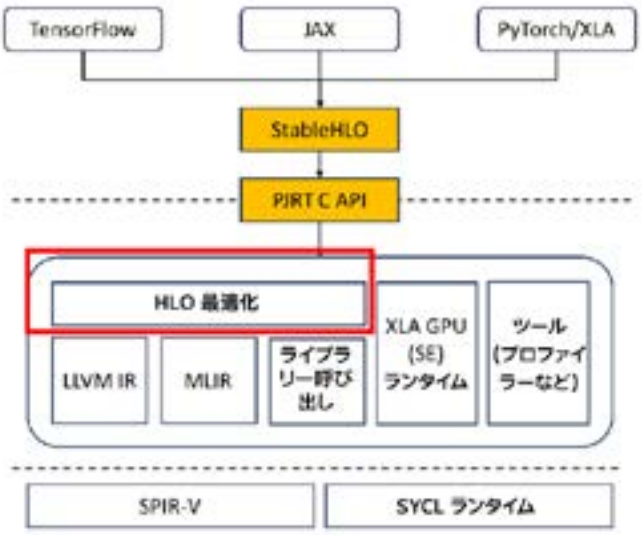
JIT コンパイル・コンテキスト：すべてのコードスニペットは JIT コンパイルされ、最適化されたコードはハードウェア上で効率的に実行されます。

関数の戻り値：最初の 2 つのスニペットの戻り値の構造は似ていますが、`jax_ir8_jit` は単一の値を返すのに対し、`jax_ir9_jit` は 2 つの値を返します。`jax_ir10_jit` の戻り値は計算結果です。

データフロー：`jax_ir8_jit` と `jax_ir9_jit` はデータ転送と出力構造であるのに対し、`jax_ir10_jit` は JAX フレームワーク内での実際の計算操作です。

`jax_ir8_jit` と `jax_ir9_jit` は、主にデータフロー検証に使用される単純なテンソル渡し関数です。`jax_ir10_jit` は、JAX フレームワーク内でのより複雑な計算操作であり、テンソル操作を組み合わせて、JAX で基本的な数学計算がどのように実行されるかを示します。

StableHLO ダンプから HLO への変換



ダンプ HLO ファイルの環境変数

```
$export XLA_FLAGS="--xla_dump_hlo_as_text --xla_dump_to=./dump"
```

すべてのパスのログをダンプするには、追加のフラグ「`--xla_dump_hlo_pass_re=.*`」が必要です。

`module_0000.jit_convert_element_type.gpu_target_config.pbtxt`（ハードウェア情報）
`module_0000.jit_convert_element_type.before_optimizations.txt`（初期 HLO）
`module_0000.jit_convert_element_type.sm_8.0_gpu_after_optimizations.txt`（最終 HLO）

図 3. StableHLO ダンプから HLO への変換に関連するファイルの一覧

JAX 表現を StableHLO にダンプするフェーズでは、各ファイルが OpenXLA への入力として使用され、対応する HLO 表現のステップが生成されます。上記は `jit_integer_pow` モジュールの詳細を示しており、ほかのステップを理解する参考資料として使用できます。詳細は、OpenXLA 公式ドキュメント(英語)を参照してください。

ファイル分析

- ハードウェア情報ファイル（`module_0000.jit_convert_element_type.gpu_target_config.pbtxt`）

  このファイルには、`threads_per_block_limit`、`threads_per_warp`、`shared_memory_per_block`、`shared_memory_per_core` などの GPU デバイスの構成やパフォーマンス・パラメーターを含む、使用ハードウェアに関する静的な情報が記録されます。

- 初期 HLO（`module_0000.jit_convert_element_type.before_optimizations.txt`）

  `jit_integer_pow` という HLO モジュールを定義します。このモジュールは、入力されたスカラー浮動小数点数を 2 乗する計算プロセスを記述します。これは、前出の JAX 表現のダンプの x の 2 乗を表す `integer_pow` 演算に直接対応しており、`Arg_0.1` を自身で乗算する HLO 表現に変換されています。乗算は入力パラメーターの 2 乗を計算します。

  `HloModule jit_integer_pow`：これは `jit_integer_pow` という名前の HLO モジュールです。HLO モジュールは、XLA コンパイラーによって使用される中間表現であり、通常はディープラーニング計算グラフの最適化と実行に使用されます。

  `entry_computation_layout={(f32[])->f32[]}`:この部分は、エントリー計算のレイアウトを定義します。具体的には、計算グラフが `f32[]` スカラー浮動小数点数を入力として受け取り、`f32[]` スカラー浮動小数点数を出力として返すことを記述します。

  `metadata={...}`：この部分は、操作にメタデータをアタッチします。メタデータは、操作名、ファイルの場所、ソースファイル内の操作の位置など、ソースコード内の操作のソースの追跡に使用されます。

  `source_line=*`：この操作がソースファイルの `*` 行目で定義されていることを示します。

- 最終 HLO

  HLO モジュール `jit_integer_pow` には、2 つの計算が含まれています。

  - `wrapped_multiply_computation`：`param_0` の 2 乗を計算します。
  - `main.3`：エントリーポイントを定義し、入力パラメーター `Arg_0.1.0` を `wrapped_multiply_computation` に渡して計算を行い、結果を返します。

  このモジュールは、SPMD（Single Program Multiple Data）共有伝播を設定し、計算レイアウトを指定し、フロントエンド属性とメタデータをアタッチします。

  `is_scheduled=true`：計算グラフがスケジュールされていることを示します。つまり、演算が特定の順序で実行されることを意味します。

StableHLO と HLO

- **変換チェーン：** StableHLO は高水準モデル計算に使用される初期ステージの表現であり、HLO は後続ステージを表します。StableHLO は HLO に変換され、さらに最適化されてマシンコードが生成されます。
- **最適化ブリッジ：** StableHLO は安定したインターフェイスを提供し、HLO はこれらの高水準抽象化をハードウェア実行向けに最適化します。この 2 つの間の変換は、OpenXLA コンパイラー・チェーンの重要な部分です。

StableHLO は表現と抽象化に重点を置いた高水準表現であり、HLO はこれらの表現をさらに改良し、計算グラフを最適化してターゲット・ハードウェアへの効率的なマッピングを実現します。

まとめ

パート 1 では、JAX フレームワークが Python 表現を StableHLO 表現に変換する方法や、StableHLO ダンプを HLO に変換する方法など、JAX フレームワークと OpenXLA* 向けインテル® エクステンションの基本概念を紹介しました。StableHLO と HLO に関連する各ファイルの概要も説明しています。**パート 2** では、HLO ダンプから LLVM 中間表現を作成し、最終的にインテルのGPU 向けに実行可能な SPIR-V ファイルを生成する実行ワークフローを説明します。

intel. tiber™ AI Cloud

インテルの AI クラウドで AI 革新を加速

最先端のツールとインテルの最適化されたソフトウェアとハードウェアを活用して、エンタープライズ、クラウド、HPC など、あらゆる環境で AI ソリューションを開発、最適化、展開できます。

今すぐ開始

JAX と OpenXLA の実行プロセスと基本ロジック

パート 2

Zhenming Wang、Lu Teng、Yabai Hu、Yi Zhou、Lifeng Wang、Yi Yao、Jack Chen、Xuming Gai、Wenjun Liu インテル コーポレーション AI フレームワーク チーム

パート 1 では、JAX が Python 表現を StableHLO 表現に変換する方法や、StableHLO ダンプから HLO（高水準演算）への変換など、JAX フレームワーク（英語）と OpenXLA* 向けインテル® エクステンション（英語）の基本概念について説明しました。パート 2 は実行ワークフローの続きで、HLO ダンプから LLVM IR（中間表現）を作成し、最終的にインテルの GPU で実行可能な SPIR-V ファイルを生成します。

実際の実行ロジック

HLO から LLVM IR への変換

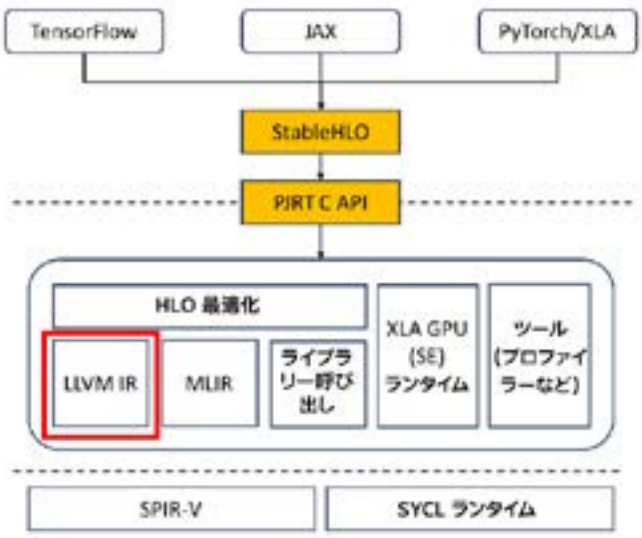


図 1. PJRT プラグインが実装された OpenXLA* 向けインテル® エクステンション(出典(英語))

HLO 表現から LLVM 中間表現（IR）への変換において、インテルの OpenXLA コンパイラーは、高水準演算表現から低水準マシンコードを生成します。HLO はテンソル演算を基本単位として使用し、モデルに含まれる数学演算とテンソル操作を表現します。詳細は、LLVM ドキュメント（英語）を参照してください。

HLO は、OpenXLA コンパイラーで使用される高水準中間表現であり、ディープラーニング・モデルの計算グラフを表します。LLVM IR は、マシンコード抽象化に近い低水準中間表現であり、通常は特定の命令レベルの演算の記述に使用されます。LLVM IR は LLVM コンパイラー・フレームワークの中核であり、さまざまなコンパイラー・ツールチェーンで広く使用されています。

HLO と LLVM IR

- **変換プロセス：** OpenXLA のコンパイルフローにおいて、HLO は高水準の中間表現です。一連の最適化を経て、最終的に LLVM IR 表現に変換されます。この変換プロセスでは通常、HLO の高水準演算を改良し、それらを低水準の命令セットにマッピングします。
- **マッピングと改良：**
  - **演算マッピング：** 各 HLO 演算は、複数の LLVM IR 命令に対応する場合があります。例えば、行列乗算の HLO 演算は、LLVM IR の一連のロード命令、乗算命令、累算命令に展開される場合があります。
  - **ハードウェア固有の最適化：** LLVM IR の生成中に、コンパイラーは特定のハードウェア機能（ベクトル化、パイプライン化など）を活用して、さらなる最適化を行い、効率良いコードを生成する場合があります。

- **コード生成 :** HLO が LLVM IR に変換されると、LLVM IR は LLVM バックエンド・ツールチェーンを介して処理され、最終的にハードウェア・プラットフォーム固有のマシンコード（x86、Arm、CUDA コードなど）が生成されます。ただし、インテルの GPU では最初に SPIR-V に変換されます。このマシンコードはターゲットデバイス上で実行されます。
- **HLO と LLVM IR の関係性 :** HLO と LLVM IR の関係は、高水準テンソル計算表現から低水準マシンコード生成への橋渡しと見ることができます。HLO は、プラットフォームに依存しない高水準抽象化を提供し、グローバル最適化を容易にします。一方、LLVM IR はこれらの抽象化を特定の命令セットに変換し、ハードウェア・レベルの最適化と実行を可能にします。OpenXLA コンパイラーはこのプロセスにより、ディープラーニング・モデルをさまざまなハードウェア・プラットフォームに効率的にマッピングし、最適な計算性能を実現します。

ダンプ LLVM ファイルの環境変数

すべてのパスのログをダンプするには、「`--xla_dump_hlo_pass_re=.*`」オプションを追加する必要があります。

```
$export XLA_FLAGS="--xla_dump_hlo_as_text --xla_dump_to=./dump"
```

```
module_0001.jit_integer_pow.ir-no-opt.ll （初期 LLVM IR）
module_0000.jit_convert_element_type.ir-with-opt.ll （最終 LLVM IR）
module_0001.jit_integer_pow.spv （SPIR-V ファイル）
module_0001.jit_integer_pow.thunk_sequence.txt （実行順序）
```

図 2. HLO ダンプから LLVM への変換に関連するファイルの一覧

ファイル分析

- 初期 LLVM IR

  インテルの OpenXLA コード生成機能により、HLO から LLVM に変換します。

  SPIR アーキテクチャーの OpenCL/CUDA カーネルでは、LLVM IR を使用して記述されます。GPU 上で乗算を実行するシンプルなカーネル関数（`wrapped_multiply`）を定義します。

  `target datalayout` : この文字列は、LLVM IR のデータレイアウトを指定します。ビット幅、アライメント、各種整数型、浮動小数点型、ベクトル型の詳細など、メモリー内でのデータ配置方法をコンパイラーに指示します。

  Target triple : ターゲット・プラットフォーム情報を指定します。

  このコードは、ワークグループ ID とスレッド ID の取得やシンプルな乗算など、いくつかの予備計算を実行する `spir_kernel` カーネル関数 `wrapped_multiply` を定義します。

  SPIR-V 組込み関数呼び出し : `__spirv_BuiltInWorkgroupId` や `__spirv_BuiltInLocalInvocationId` などの SPIR-V 組込み関数を呼び出して、GPU 並列計算に不可欠なスレッド ID 情報を取得します。

- 最終 LLVM IR

  最適化と変換を経た最終 LLVM IR は、最終的に生成されるマシンコードの形式に近くなります。これには、初期 IR に加えて、追加の最適化とハードウェア固有の変換が適用されています。

  - **カーネルエントリー関数 :** `wrapped_multiply` に加えて、追加のカーネルエントリー関数 `__spirv_entry_wrapped_multiply` が定義されています。この関数は、`wrapped_multiply` を呼び出し、適切なカーネル実行環境の確保に必要な追加のカプセル化を提供します。

  - **ハードウェア・レベルの最適化:** コードには、マシンコード生成の準備として、レジスター割り当て、命令最適化、アライメント修正などの最適化が適用されています。

  - **定数組込み関数 :** メモリーアドレス空間 8 の `@__spirv_BuiltInWorkgroupId` や `@__spirv_BuiltInLocalInvocationId` などの定数は、ハードウェア関連の最適化を反映した特定のデータ保存場所を示します。

初期 LLVM IR と最終 LLVM IR の違い

- **構造と表現：**
  - **初期 LLVM IR：**より抽象的で、高水準演算とプラットフォーム非依存のロジックを含みます。このステージの IR は、主に論理的な正確さと抽象的な表現を重視して、計算プロセスが記述されています。
  - **最終 LLVM IR：**より具体的でハードウェア層に近く、さらなる最適化とハードウェア・マッピングが行われています。この IR は、ターゲット・アーキテクチャーに特化した命令生成とレジスター割り当ての最適化が適用されています。
- **呼び出しと定義の変更：**
  - **組込み関数の扱い：**初期の LLVM IR は SPIR-V 組込み関数を直接呼び出していましたが、最終 LLVM IR ではこれらの関数は定数ロードに置き換えられています。関数定義はより具体的になり、実際のハードウェア・アクセス・パターンを反映しています。
  - **エントリー関数：**カーネルの正しい実行および初期化環境を確保するため、最終 LLVM IR にはエントリー関数 `__spirv_entry_wrapped_multiply` が追加されています。

初期 LLVM IR と最終 LLVM IR の関係性

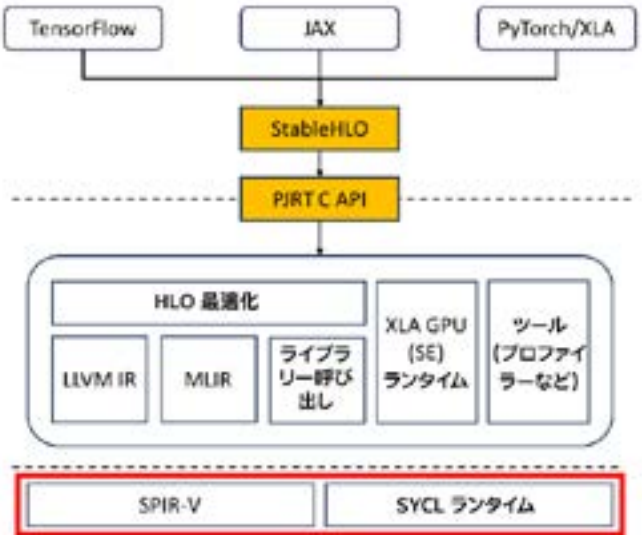
- **変換と最適化：**
  - **初期から最終へ：**コンパイラーは初期 IR から開始し、定数伝播、レジスター割り当て、命令選択、メモリー・アクセス最適化などの一連の最適化を適用して、より効率的で具体的な最終 IR を生成します。
  - **増分改良：**初期 IR はコンパイルプロセスにおける中間状態として機能し、演算の高水準抽象化を維持します。最終 IR の生成中に、コンパイラーはこれらの抽象化を改良し、実際のハードウェア命令にマッピングします。
  - 初期 HLO と最終 HLO 間の最適化は OpenXLA* 向けインテル® エクステンションによって行われ、初期 LLVM IR と最終 LLVM IR 間の最適化は OpenXLA* 向けインテル® エクステンションではなく LLVM によって行われます。
- **機能的一貫性：**

  形式には違いがあるにもかかわらず、2 つのバージョンの IR は機能的には一致しています。最終 IR は初期 IR の計算ロジックを保持していますが、効率性を重視して最適化され、ターゲット・ハードウェアの特定の要件を満たすように調整されています。

初期 LLVM IR はコンパイルプロセスの初期ステージの表現であり、より抽象的で、計算ロジックと基本操作の記述に重点を置いています。一方、最終 LLVM IR は最適化された表現であり、ハードウェア固有の最適化と調整を経て、最終的なマシンコードに近いものとなっています。

これら 2 つの IR ステージは、高水準言語コードを低水準マシンコードに変換するプロセスの異なるステージを表しています。これらのステージ間の変換と最適化は、最終的に生成されるコードの実行効率とパフォーマンスに直接影響します。

SPIR-V ファイル



SPIR-V は、Standard Portable Intermediate Representation（SPIR）のバイナリー形式バージョンであり、GPU やその他のアクセラレーターで実行される計算タスクのコンパイル済みバイナリーコードが含まれています。

LLVM IR をインテルの GPU 向けの SPIR-V に変換するトランスレーターがあります。詳細は、GitHub リポジトリー（英語）を参照してください。これは、並列コンピューティングおよびグラフィックス・レンダリング・タスクの記述に使用されるバイナリー中間表現形式です。SPIR-V は、さまざまなプラットフォームに統一された表現を提供することで、開発者がさまざまなヘテロジニアス・コンピューティング環境で効率良くコードを実行できるようにします。

生成された SPIR-V ファイルは、ターゲット・ハードウェアに直接ロードして実行できるため、デバイスの計算能力を最大限に活用できます。

Python 表現による `x^2`（x の 2 乗計算）は、JAX フレームワークによって処理され StableHLO 表現が生成された後、OpenXLA によって HLO に変換され、さらに LLVM IR に変換されます。これらの LLVM IR 表現は、一連の最適化と変換の手順を経て、最終的に SPIR-V バイナリーファイルを生成します。SPIR-V ファイルは、ターゲット・ハードウェア上で効率的に実行できます。

実行のため、ファイルを GPU に送信する手順は次のとおりです。

1. **生成されたバイナリーファイルのロード :** バイナリーファイルがロードされ、さまざまな API 呼び出しと変換を経て、SYCL が認識可能なカーネルが生成されます。
2. **並列属性の追加 :** さまざまな並列属性が追加され、`sycl_nd_range` が作成されます。
3. **並列実行の開始:**`parallel_for` は、生成されたカーネル、データポインター、および計算に関連するパラメーターを含む並列実行のスケジュールを開始します。
4. **キューへの送信 :** `queue.submit` は、実行のため CPU から GPU にタスクを分配します。そして、インテルの GPU コンパイラーが、GPU が認識して計算に使用できるファイルを生成します。

プログラムの実行が完了するまで、このプロセスが継続されます。

```
auto event = queue.submit([&](sycl::handler &cgh) {
cgh.set_arg(0, ptr0);
cgh.set_arg(1, ptr1);
cgh.set_arg(2, ptr2);
cgh.set_arg(3, ptr3);
cgh.parallel_for(sycl_nd_range, kernel);
});
```

まとめ

パート 2 では、OpenXLA 上で実際のケースを実行する具体的な実行プロセスと基盤となるロジックを紹介しました。OpenXLA は、StableHLO 表現を HLO および LLVM 中間表現に変換し、最終的にインテルの GPU 向けに実行可能な SPIR-V ファイルを生成し、その後、再構成とパッケージ化を行います。この記事は、インテルの GPU 向けの OpenXLA コンパイラー開発と JAX/OpenXLA のデバッグに取り組む開発者に初期考察を提供するものです。

intel software

THE PARALLEL UNIVERSE