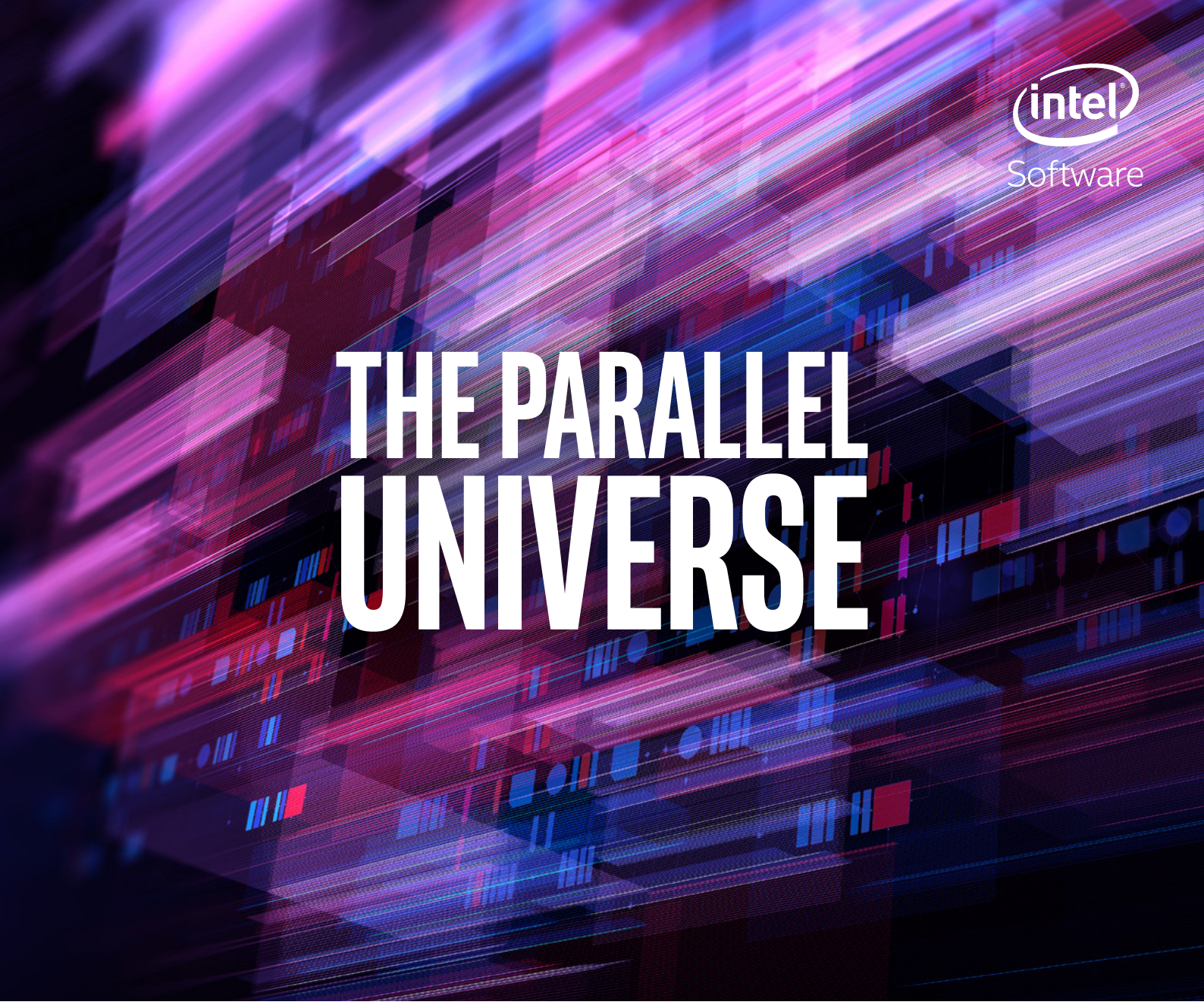


BigDL により Apache Spark* 上で人工知能を向上

WebAssembly が Web 上のコンピューティングの未来を握る理由

コードの現代化を実践 : スレッド化、メモリー、ベクトル化の最適化

Issue
33
2018

目次

編集者からのメッセージ 3
BigDL が注目を集めている理由
Henry A. Gabb インテル コーポレーション シニア主席エンジニア

注目記事

BigDL により Apache Spark* 上で人工知能を向上 5
機能、ユースケース、今後の展開

WebAssembly が Web 上のコンピューティングの未来を握る理由 19
ブラウザーでの処理の歴史と新たな動向

コードの現代化を実践 : スレッド化、メモリー、ベクトル化の最適化 29
最新のインテル® ソフトウェア開発ツールによりハードウェアをさらに効率良く使用

Java* による不揮発性メモリー・コンピューティング 43
現代のアプリケーション・プログラミングにおける適応性の鍵

より高速な勾配ブースティング決定木 55
インテル® DAAL によりマシンラーニングのパフォーマンスを向上する方法

MPI-3 の非ブロッキング集合操作による通信レイテンシーの隠蔽 63
通信と計算のオーバーラップにより HPC パフォーマンスを向上

コンパイラーの最適化に関する詳細は、**最適化に関する注意事項**を参照してください。

編集者からのメッセージ

Henry A. Gabb インテル コーポレーション シニア主席エンジニア
HPC と並列コンピューティング分野で長年の経験があり、並列プログラミングに関する記事を多数出版しています。『Developing Multithreaded Applications: A Platform Consistent Approach』の編集者 / 共著者で、インテルと Microsoft* による Universal Parallel Computing Research Centers のプログラム・マネージャーを務めました。

BigDL が注目を集めている理由

私が大規模データ解析に Apache Spark* を使い始めた頃、MLlib はこのフレームワークでマシンラーニングを行うための唯一の選択肢でした。MLlib は優れたパッケージですが、制限もあります。BigDL は Apache Spark* にディープラーニングをもたらしました。2016 年 12 月に BigDL が最初にリリースされたとき、私は開発者にその概要を尋ねました : 「**BigDL: Apache Spark* 上の最適化されたディープラーニング**」(The Parallel Universe 28 号)。この号の注目記事「**BigDL により Apache Spark* 上で人工知能を向上**」では、それから 1 年半経った BigDL の新機能をチェックします。

お気に入りのプログラミング言語でアプリケーションを開発し、Web に配備して、ネイティブに近いパフォーマンスを達成できることを想像してみてください。前号では、Web ブラウザー内で高度なコンピューター・ビジョン計算を行う新しいテクノロジーである OpenCV.js を紹介しました : 「**コンピューター・ビジョンの普及**」(The Parallel Universe 32 号)。この号の記事「**WebAssembly が Web 上のコンピューティングの未来を握る理由**」では、ブラウザー内で複雑な計算を実行する新しいテクノロジーを紹介します。

私は、アプリケーションのパフォーマンス・データを収集して、簡潔に表示するプログラミング・ツールを評価したことがあります。最初は、その豊富なデータとカラフルな GUI に魅了されました。しかし、目新しさがなくなると、どのデータも実用的ではなく、アプリケーションのチューニングにそれほど有用ではないことに気付きました。「**コードの現代化を実践**」では、インテル® Parallel Studio XE の解析ツールを利用してコードの現代化を行う方法を示します。

不揮発性メモリーは、ますます重要なハードウェア・テクノロジーになっています。「**Java* による不揮発性メモリー・コンピューティング**」では、Java* アプリケーションでこのテクノロジーを利用できるようにするライブラリーを紹介します。不揮発性メモリーについては、今後の The Parallel Universe でも取り上げていく予定です。

コンパイラーの最適化に関する詳細は、**最適化に関する注意事項**を参照してください。

「**より高速な勾配ブースティング決定木**」では、インテル® データ・アナリティクス・アクセラレーション・ライブラリー (インテル® DAAL) の最新バージョンの新機能と拡張を紹介します。

インテル® MPI ライブラリーも、新しいハードウェアと MPI 規格の変更に対応するため継続的に開発が進められています。私の知る限り、MPI のレイテンシーを隠蔽するため常に通信と計算をオーバーラップすることは、一般的な手法とされてきました。しかし、実際のアプリケーションで通信と計算のオーバーラップによるパフォーマンスの利点を示すことは困難です。そのため、数年前に MPI 規格に非ブロッキング集合操作が追加されたとき、私はそれほど注目していませんでした。「**MPI-3 の非ブロッキング集合操作による通信レイテンシーの隠蔽**」は、これらの新しい関数について、特に大規模なデータセットを処理する多数の計算ノードを利用する必要がある場合に再考するきっかけとなりました。

今後の The Parallel Universe では、ディープラーニング向けの FPGA、Python* によるスレッド化、大規模な分散データ解析に対する新しいアプローチ、インテル® ソフトウェア・ツールの新機能などに関する記事をお届けします。お見逃しなく。

Henry A. Gabb
2018 年 7 月

コンパイラーの最適化に関する詳細は、**最適化に関する注意事項**を参照してください。

BigDL により Apache Spark* 上で人工知能を向上

機能、ユースケース、今後の展開

Jason Dai インテル コーポレーション シニア主任エンジニア
Radhika Rangarajan インテル コーポレーション エンジニアリング・マネージャー

インテルが 2016 年 12 月に発表して以来、**BigDL** (英語) は活発なオープンソース・プロジェクトに発展しました。この記事では、BigDL の実装について詳しく説明し、いくつかの実践的なユースケースを紹介します。また、新しいエンドツーエンド解析、Apache Spark* と BigDL ベースの人工知能 (AI) パイプライン (**Analytics Zoo** (英語) プラットフォーム) の概要を説明します。[**編集者注 :** このフレームワークの最初のバージョンについては、The Parallel Universe 28 号の「**BigDL: Apache Spark* 上の最適化されたディープラーニング**」で取り上げました。それ以来、多くの変更がありました。]

BigDL を利用する理由

ビッグデータ・プラットフォームと解析パイプラインにディープラーニング・テクノロジー (コンピューター・ビジョン、自然言語処理、GAN (Generative Adversarial Network)) を適用したいと考える企業が増えています。ディープラーニングとビッグデータ解析の収束をサポートするため、インテルは 2016 年 12 月に BigDL 分散ディープラーニング・フレームワークをオープンソース化しました。

BigDL は、Apache Spark* ライブラリーとして実装されます。大規模なディープラーニング・アプリケーション (モデルの訓練と推論を含む) を既存のビッグデータ・クラスター (Hadoop* または Apache Spark*) 上で直接実行可能な標準の Apache Spark* プログラムとして記述できます。

BigDL は、ディープラーニング・テクノロジーを包括的にサポートします。

- ニューラル・ネットワーク操作
- 層
- 損失
- オプティマイザー

特に、ほかのフレームワーク (TensorFlow*、Keras、Caffe*、Torch) で定義された既存のモデルを分散して Apache Spark* 上で直接実行できます。また、BigDL はディープラーニング・テクノロジーとビッグデータ・エコシステムのシームレスな統合を可能にします。BigDL プログラムは、Apache Spark* のコンポーネント (DataFrame、Spark Streaming、ML Pipeline) と直接やり取りできるだけでなく、さまざまなビッグデータ・フレームワーク (Apache Storm*、Apache Flink*、Apache Kafka* など) で動作します。

BigDL アプリケーションの記述

BigDL は Apache Spark* で実装されているため、次の 5 つのステップに従って、エンドツーエンドのデータ解析を容易に行うことができます。

ステップ 1: テキストデータ (項目と関連ラベル) を Apache Spark* の RDD (Resilient Distributed Dataset) に読み込みます。

```
spark = SparkContext(appName="text_classifier", …)
texts_rdd = spark.textFile("hdfs://...")
words_rdd = texts_rdd.map(lambda text, label:
                    ([w for w in to_words(text)], label))
```

ステップ 2: RDD 変換機能によりロードしたデータを訓練サンプルの RDD に変換します。

```
w2v = news20.get_glove_w2v(dim=…)
vector_rdd = words_rdd.map(lambda word_list, label
                ([to_vec(w, w2v) for w in word_list], label))
sample_rdd = vector_rdd.map(lambda vector_list, label:
                              to_sample(vector_list, label))
```

ステップ 3: ニューラル・ネットワーク・モデルを構築します。

```
model = Sequential().add(Recurrent().add(LSTM(…)))
                    .add(Linear(…))
                    .add(LogSoftMax())
```

ステップ 4: (指定した損失と最適化手法で) モデルを訓練します。

```
loss = ClassNLLCriterion()
optim_method = Adagrad()
optimizer = Optimizer(model=model, training_rdd=sample_rdd,
                criterion=loss, optim_method=optim_method, …)
optimizer.set_train_summary(summary=TrainSummary(…))
trained_model = optimizer.optimize()
```

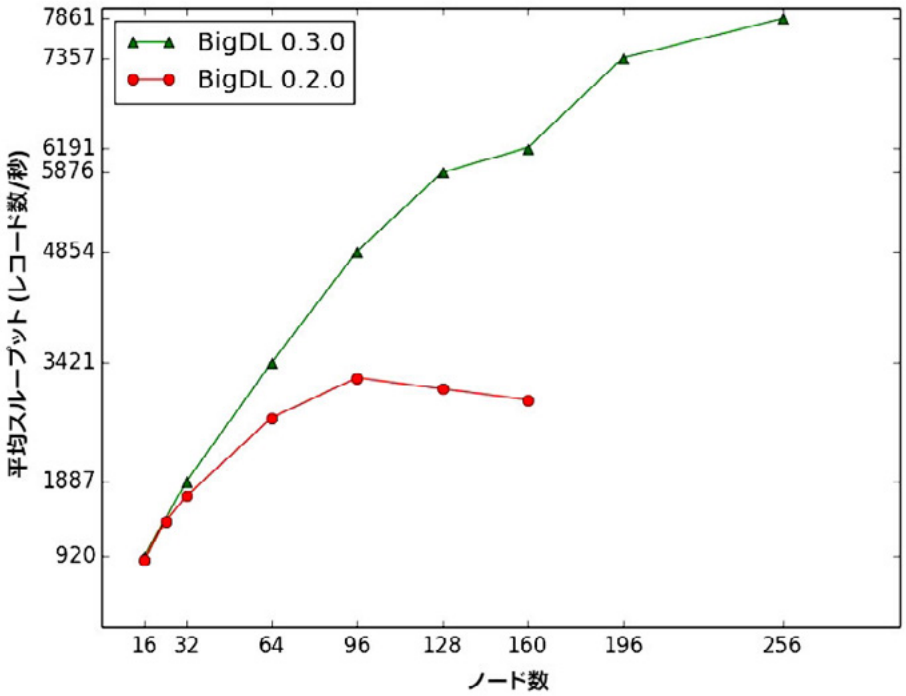
ステップ 5: 既存のモデル (BigDL、TensorFlow*、Keras、Caffe*、または Torch で訓練済み) を使用して結果を予測します。

```
test_rdd = …
prediction_rdd = trained_model.predict(test_rdd)
```

分散訓練

BigDL は、同期ミニバッチ確率的勾配降下法とデータ並列処理を利用して極めてスケーラブルな分散訓練を提供します。(all-reduce 操作を利用して) Apache Spark* 内部の BlockManager 上に直接パラメーター・サーバー (PS) アーキテクチャーを実装します。各タスクはローカルの勾配を計算した後、結果をドライバーに戻す代わりに、単一のワーカー内のすべてのパーティションの勾配をローカルに集計します。そして、各ノードの集計された勾配は、すべてのクラスターノード間で交換されるチャンクにスライスされます。各ノードは特定のチャンクを担当します。これには、パラメーター同期を行う BigDL の PS アーキテクチャーが実装されます。

各ノードは、計算を担当するモデルのスライスの勾配を取得および集計します。ペアの交換が完了した後、各ノードはそれぞれ集計済みの勾配を使用して重みを更新します。そして、更新された重みを同期するため再度交換が行われます。この処理の最後に、各ノードは更新されたコピーを持ちます。これにより、BigDL は高度にスケーラブルなディープラーニング・モデルの分散訓練を実現できます (**図 1**)。



1 **Cray* により報告された BigDL 0.3.0 とデュアルソケット インテル® Xeon® プロセッサー E5-2620 ベースのノードを使用した ImageNet Inception v1 訓練のスループット。[1] スループットは最大 128 ノードまでほぼ直線的にスケーリングし、256 ノードまでスケーリングを続けています。**

モデルの量子化

量子化とは、数値を格納し、よりコンパクトで低精度の計算を実行するテクノロジーを指します。BigDL は、このような低精度の計算を利用して、最適化された推論向けに既存のモデルを量子化します。BigDL は最初に既存のモデルをロードし、選択された層 (例 : 空間畳み込み) のパラメーターを 8 ビット整数形式に量子化して、量子化されたモデルを生成します (**図 2**)。

モデルの推論中、量子化された層はそれぞれ次のことを行います。

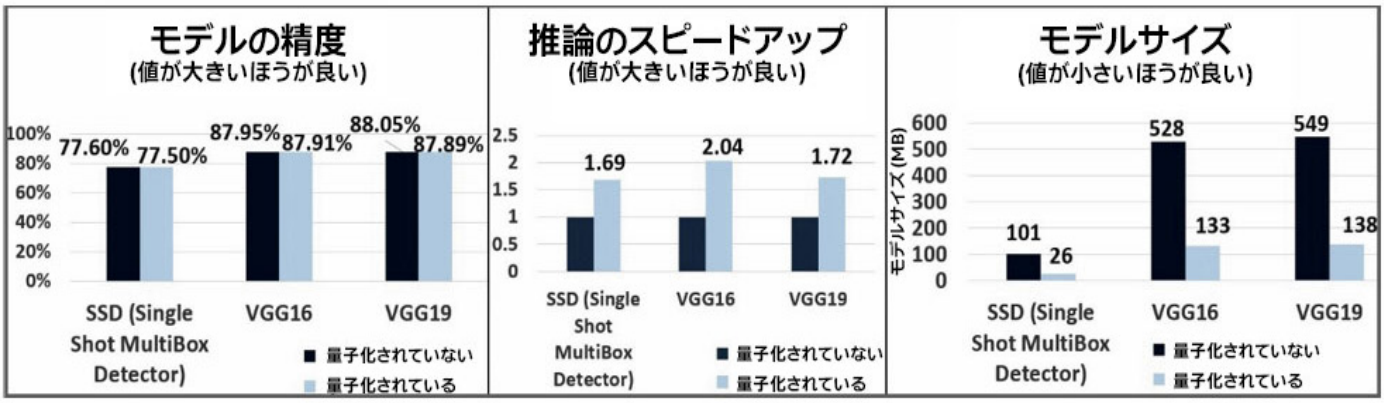
- **量子化 :** 32 ビット浮動小数点数の入力値を 8 ビット整数に量子化します。
- **実行 :** 量子化されたパラメーターとデータを使用して 8 ビットの計算 (GEMM など) を実行します。
- **逆量子化 :** 結果を 32 ビット浮動小数点数に逆量子化します。

これらの操作の多くは実装で融合できるため、推論時の量子化と逆量子化のオーバーヘッドは非常に低くなります。

```
Math.round(1.0 * value
           / Math.max(Math.abs(max), Math.abs(min))
           * Byte.MaxValue).toByte
```

2 **32 ビット浮動小数点数から 8 ビット整数への量子化の式**

多くの量子化の実装とは異なり、BigDL は小さなローカルの量子化ウィンドウ (パラメーターや入力データの畳み込みのパッチやカーネルなどの小さなサブブロック) ごとに量子化と逆量子化を実行する、新しいローカルの量子化スキームを採用しています。そのため、BigDL はモデルの量子化においてビット数が非常に小さい整数 (8 ビット) を使用し、精度の低下を最小限 (0.1% 未満) に抑え、モデルサイズを 1/4 に減らし、推論で 2 倍のスピードアップを達成できます (**図 3**)。



3 **BigDL 0.3.0 と Amazon Web Services* EC2 C5.18xlarge インスタンス[2] を使用した SSD、VGG16、VGG19 のモデル量子化の結果 (精度、推論のスピード、モデルサイズ)**

ローカル実行

標準の Apache Spark* プログラムであることに加えて、BigDL はローカル Java* 仮想マシン (JVM) 上で Apache Spark* なしでモデルの訓練と推論をサポートします。BigDL を単一ノード上で実行する場合、パラメーター同期やタスク・スケジューリングのオーバーヘッドが発生しないため、効率の向上につながります。さらに重要なのは、推論や微調整のために BigDL モデルをさまざまな JVM ベースのビッグデータ・フレームワーク (Apache Storm*、Apache Flink*、Apache Kafka* など) と容易に統合できることです。

ユースケースと応用

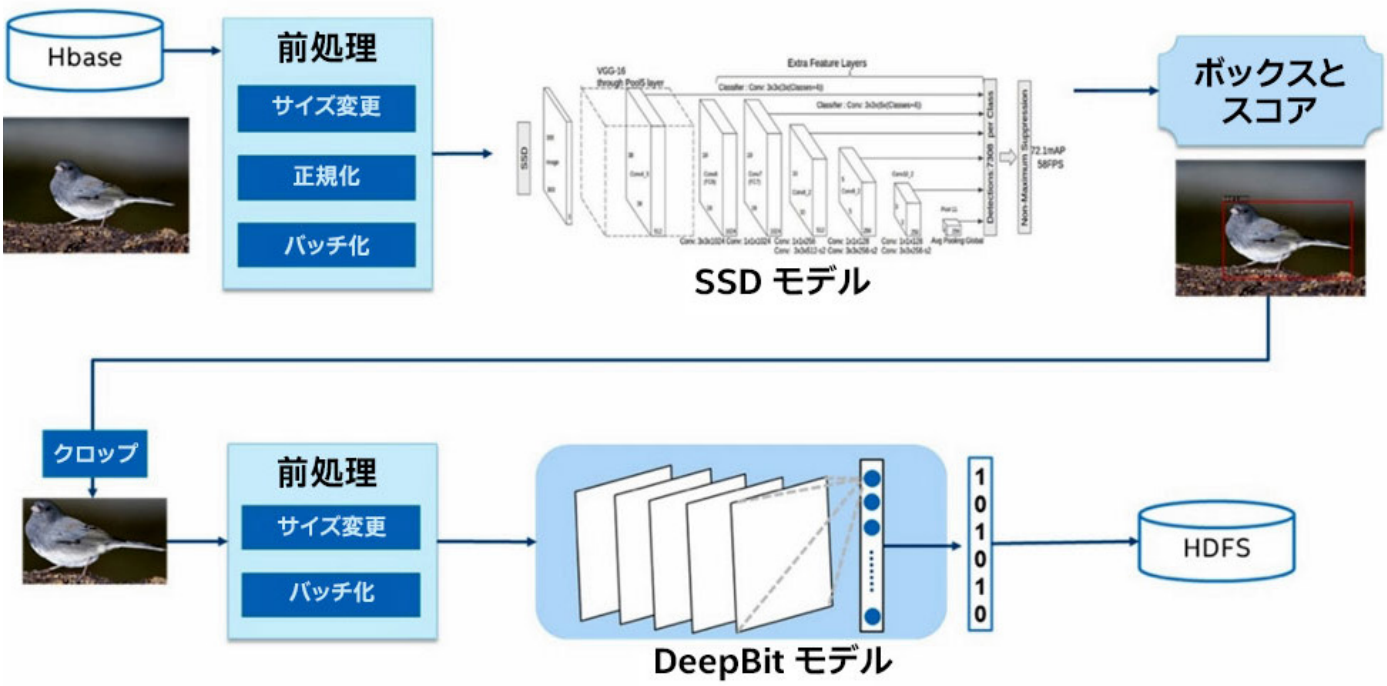
このセクションでは、3 つの典型的な BigDL のユースケースを示します。

1. モデルの推論
2. 分散訓練
3. 転移学習

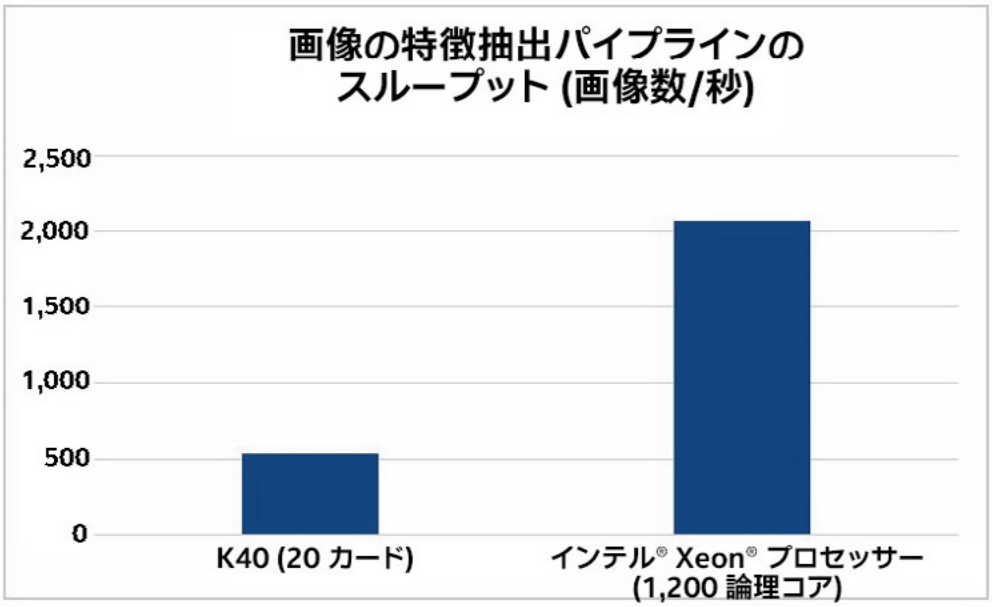
画像の特徴抽出

世界最大のオンライン小売業者の 1 つである JD.com (中国語) は、BigDL と Apache Spark* ベースのエンドツーエンドの物体検出と画像の特徴抽出パイプラインを構築しました (**図 4**)。パイプラインは最初に、分散データベースから Apache Spark* へ数百万枚の画像を (画像の RDD として) 読み込み、Apache Spark* を使用して分散方式で RDD を前処理します (サイズ変更、正規化、バッチ化を含む)。その後、BigDL を使用して SSD モデル (Caffe* で訓練済み) をロードし、Apache Spark* 上で大規模な分散物体検出を行って、各画像で検出された物体の座標とスコアを生成します。次に、ターゲット画像を生成して (最高スコアの物体をターゲットとして、その座標を基にオリジナル画像を切り取って)、ターゲット画像の RDD の前処理 (サイズ変更とバッチ化) を行います。最後に、BigDL を使用して DeepBit モデル (Caffe* で訓練済み) をロードし、ターゲット画像の分散特徴抽出を行って対応する特徴を生成します。そして、結果 (抽出された物体の特徴の RDD) を Hadoop* Distributed File System (HDFS) に格納します。

Apache Spark* と BigDL を使用して、統一されたプログラミング・パラダイムの下でデータ解析とディープラーニング・パイプライン全体 (データの読み込み、分割、前処理、モデルの推論、結果の格納を含む) を簡単に実装できます。JD の報告では、エンドツーエンドのパイプラインは、GPU クラスター上で同じソリューションを実行する場合と比較して、約 3.83 倍のスピードアップをもたらしました (**図 5**)[3]。



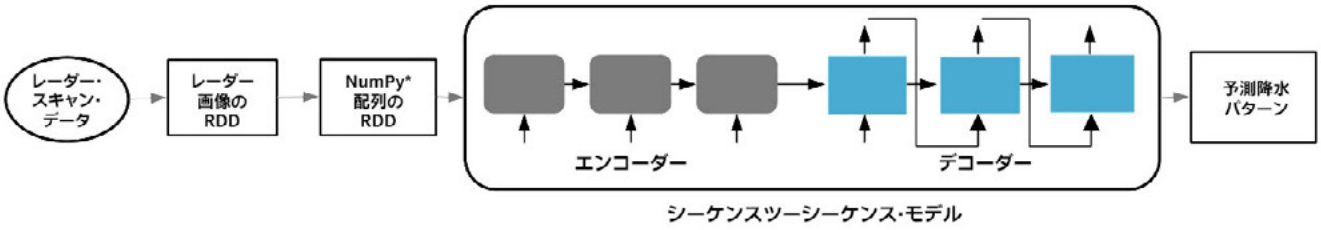
4 Apache Spark* と BigDL ベースのエンドツーエンドの物体検出と画像の特徴抽出パイプライン (SSD と DeepBit を使用)[3]



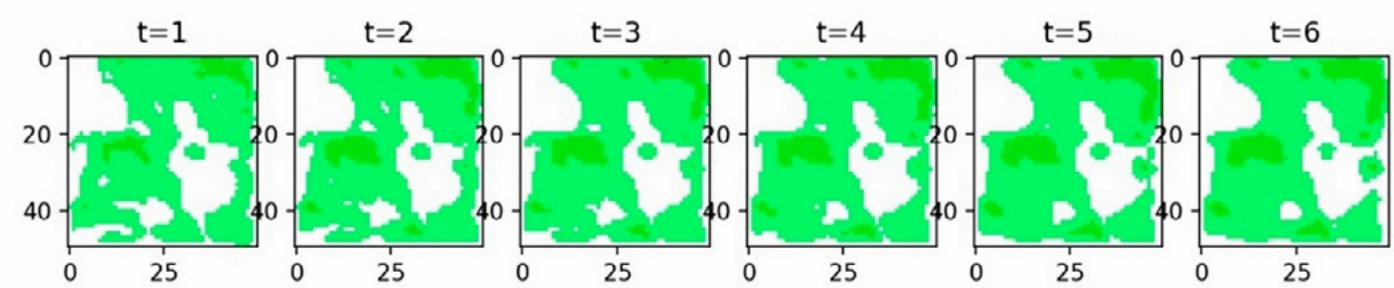
5 JD によって報告された GPU クラスターとインテル® Xeon® プロセッサー・ベースのクラスターでの画像の特徴抽出パイプライン・ベンチマークのスループット。[3] GPU スループットは 20 枚の NVIDIA* Tesla* K40 カードでテスト。インテル® Xeon® プロセッサー・ベースのクラスターのスループットは 1,200 論理コアでテスト (デュアルソケット インテル® Xeon® プロセッサー E5-2650 v4 ベースのサーバーごとに 50 論理コアを実行)。

降水ナウキャスト

Cray* (英語) は、BigDL を Urika-XC* 解析ソフトウェア・スイートに統合して、データの前処理、モデルの訓練、推論を含むエンドツーエンドの降水ナウキャスト (降水短時間予報) ワークフローを構築しました (**図 6**)。最初に、アプリケーションは 1 テラバイトを超える生のレーダー・スキャン・データをレーダー画像の RDD として Apache Spark* に読み込み、NumPy* ndarray の RDD に変換します。次に、現在の時刻までの一連の画像を入力として、シーケンスツーシーケンス・モデルを訓練し、将来の任意の時点の一連の予測画像を出力します。モデルを訓練したら、今後 1 時間の降水パターンなどの予測に使用できます (**図 7**)。



6 **Apache Spark* と BigDL 上のエンドツーエンドの降水ナウキャスト・ワークフロー**

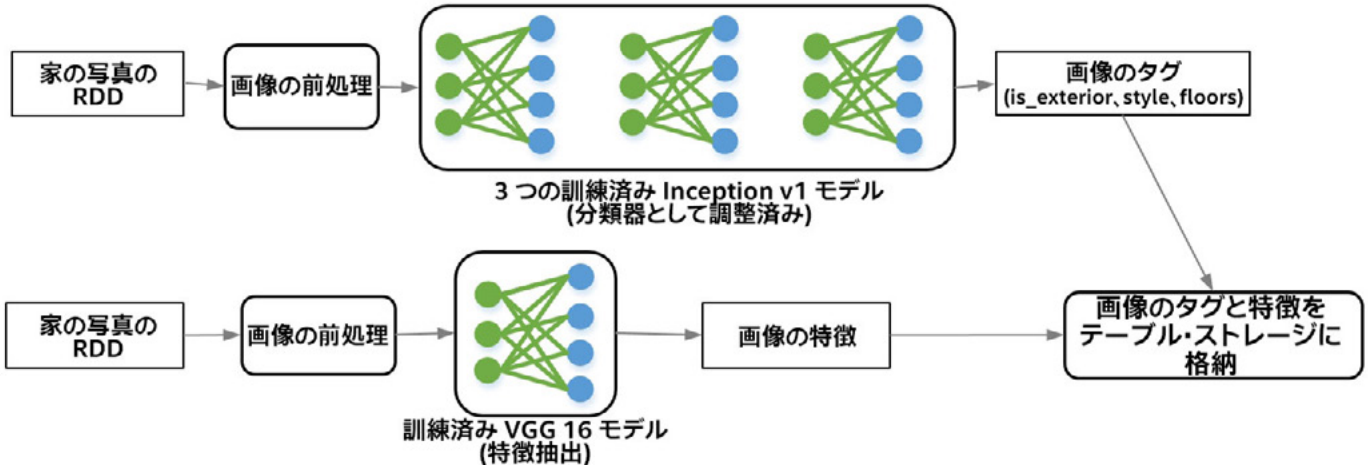


7 **Apache Spark* と BigDL を使用して今後 1 時間の降水パターンを予測**

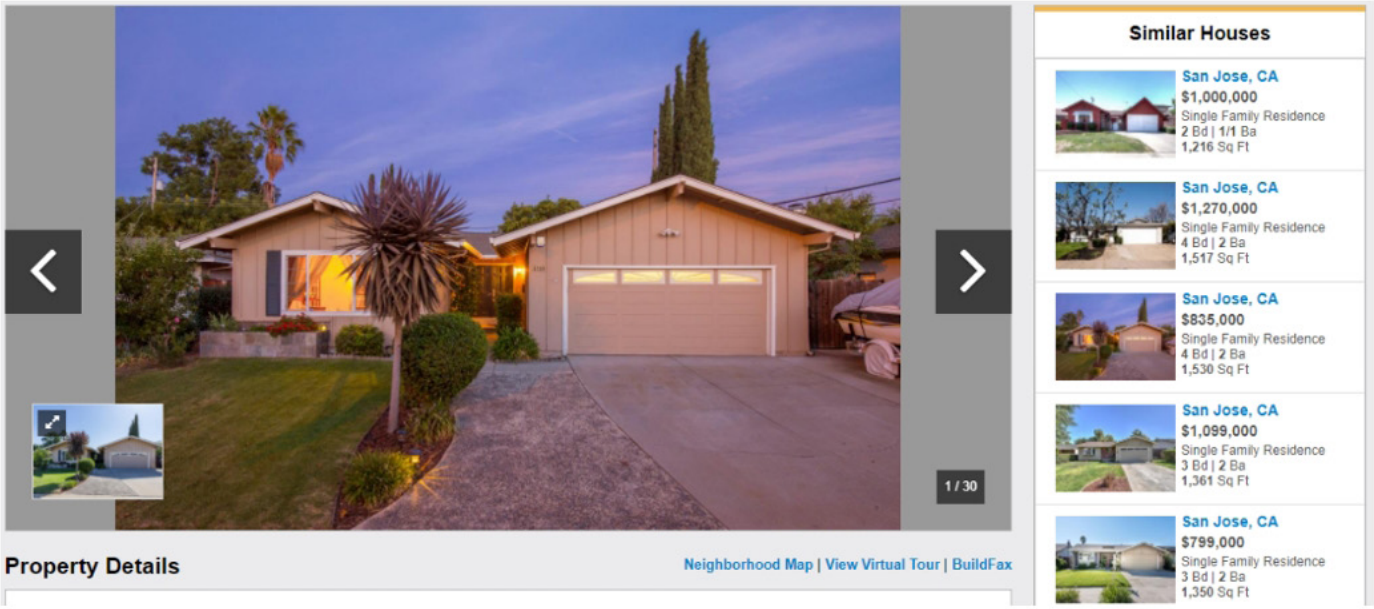
画像類似性に基づく推奨

大規模な不動産情報システム (MLS) を運営する **MLSListings Inc.** (英語) は、Apache Spark* と BigDL ベースの画像類似性に基づく物件推奨システムを構築しています。家の写真の意味的類似性と視覚的類似性の両方を計算するため、転移学習を利用してエンドツーエンドのワークフローを実装しています (**図 8**)。写真の意味的類似性を計算するため、システムは Places データセットで訓練済みの Inception v1 モデルを微調整して、[4] 写真が家の正面外観、スタイル、階数を示しているかどうかを決定する 3 つの新しい分類器を構築します。

最初に 3 つの訓練済み Inception v1 モデルをロードして、各モデルに 2 つの層 (全結合層とソフトマックス層) を追加し、新しい分類器を訓練します。訓練後、これらの分類器を使用して各物件の写真にタグ (ラベル) を付けることができます。視覚的類似性を計算するため、システムは Places データセットで訓練済みの VGG-16 モデルを使用して、各物件の写真の特徴抽出を行い、その結果を分類器によって生成されたタグと組み合わせて、分散テーブルとして格納します。モデルの実行時に、ユーザーが物件の写真を選択すると、システムは類似する視覚的特徴の物件を推奨します (**図 9**)。



8 **Apache Spark* と BigDL ベースの画像類似性に基づく物件推奨のエンドツーエンドのワークフロー**



9 **類似する物件を自動的に推奨**

BigDL の今後の展開

BigDL は Apache Spark* と統合され、ビッグデータ・アプリケーション開発者の課題に対処するため機能を拡張していますが、ライブラリーだけで実稼働環境のクラスター上で ML/DL ワークロードの配備を容易にし、高速化できるのでしょうか？ インテルは、コミュニティーのフィードバックと顧客の実装を基に、エンドツーエンド解析と AI プラットフォームである、Analytics Zoo を構築しました。Analytics Zoo は、高水準のパイプライン API、ビルトイン・ディープラーニング・モデル、リファレンス・ユースケースを提供することで Apache Spark* と BigDL アプリケーションの構築を容易にします。

高水準のパイプライン API

Analytics Zoo は、Apache Spark* と BigDL で簡単に使える、次のような高水準のパイプライン API のセットを提供します。

- **nnframes:** Apache Spark* の DataFrame と ML Pipeline のネイティブ・ディープラーニング・サポート
- **autograd:** 自動微分演算を使用したカスタム層 / 損失の構築
- **転移学習 :** 特徴抽出や微調整のための訓練済みモデルのカスタマイズ

これらの高水準のパイプライン API を使用することで、わずか数行のコードで複雑なディープラーニング・パイプラインを簡単に構築できます。

ステップ 1: NNContext を初期化し、NNImageReader を使用して DataFrame にデータをロードします。

```
from zoo.common.nncontext import *
from zoo.pipeline.nnframes import *
sc = init_nncontext ()
imageDF = NNImageReader.readImages(image_path, sc)
```

ステップ 2: DataFrame 変換を使用してデータを処理します。

```
getName = udf(lambda row: ...)
getLabel = udf(lambda name: ...)
df = imageDF.withColumn("name", getName(col("image"))) \
       .withColumn("label", getLabel(col('name')))
```

ステップ 3: ビルトインの特徴エンジニアリング操作を使用して画像を処理します。

```
from zoo.feature.image import *
transformer = RowToImageFeature() \
              -> ImageResize(64, 64) \
              -> ImageChannelNormalize(123.0, 117.0, 104.0) \
              -> ImageMatToTensor() \
              -> ImageFeatureToTensor())
```

ステップ 4: 既存のモデル (このケースでは Caffe* で訓練済み) をロードして、最後のいくつかの層を削除し、最初のいくつかの層を固定 (フリーズ) します。

```
from zoo.pipeline.api.net import *
full_model = Net.load_caffe(def_path, model_path)

# Remove layers after pool5/drop_7x7_s1
model = full_model.new_graph(["pool5/drop_7x7_s1"])

# freeze layers from input to pool4/3x3_s2 inclusive
model.freeze_up_to(["pool4/3x3_s2"])
```

ステップ 5: Keras 形式の API とカスタム Lambda 層を使用していくつかの新しい層を追加します。

```
from zoo.pipeline.api.autograd import *
from zoo.pipeline.api.keras.layers import *
from zoo.pipeline.api.keras.models import *

def add_one_func(x):
      return x + 1.0

input = Input(name="input", shape=(3, 224, 224))
inception = model.to_keras()(input)
flatten = Flatten()(inception)
lambda = Lambda(function=add_one_func)(flatten)
logits = Dense(2)(lambda)
newModel = Model(input, logits)
```

ステップ 6: Apache Spark* の ML Pipeline を使用してモデルを訓練します。

```
cls = NNClassifier(model, CrossEntropyCriterion(), transformer) \
        .setLearningRate(0.003).setBatchSize(40) \
        .setMaxEpoch(1).setFeaturesCol("image") \
        .setCachingSample(False)
nnModel = cls.fit(df)
```

ビルトイン・ディープラーニング・モデル

Analytics Zoo は、物体検出、画像分類、テキスト分類、推奨など、さまざまな問題タイプに使用できるいくつかのビルトイン・ディープラーニング・モデルを提供します。

- **物体検出 :** 物体検出 API (SSD や Faster-RCNN などの訓練済み検出モデルのセットを含む) を使用することで、物体検出アプリケーション (例えば、画像やビデオ内の複数のオブジェクトをローカライズまたは識別する) を簡単に開発できます。
- **画像分類 :** 画像分類 API (VGG、Inception、ResNet、MobileNet などの訓練済み検出モデルのセットを含む) を使用することで、画像分類アプリケーションを簡単に開発できます。
- **テキスト分類 :** テキスト分類 API は、テキスト分類アプリケーション向けの訓練済みモデル (CNN、LSTM など) のセットを提供します。
- **推奨 :** 推奨 API は、推奨エンジンを作成するため、訓練済みモデル (Neural Collaborative Filtering、Wide and Deep Learning など) のセットを提供します。

コンパイラーの最適化に関する詳細は、**最適化に関する注意事項**を参照してください。

Analytics Zoo の物体検出 API は、大きなセットの画像の推論に簡単に使用できます。

ステップ 1: Detection Model Zoo (英語)から Analytics Zoo の物体検出モデルをダウンロードします。検出モデルのコレクション (PASCAL VOC と COCO データセットで訓練済み) を利用できます。

ステップ 2: 画像データと物体検出モデルをロードします。

```
from zoo.common.nncontext import get_nncontext
from zoo.models.image.objectdetection import *

spark = init_nncontext()
image_set = ImageSet.read(img_path, spark)
model = ObjectDetector.load_model(model_path)
```

ステップ 3: オフシェルの推論と視覚化に物体検出 API を使用します。

```
output = model.predict_image_set(image_set)

visualizer = Visualizer(model.get_config().label_map(), \
                        encoding="jpg")
visualized = visualizer(output).get_image(to_chw=False) \
                               .collect()

for img_id in range(len(visualized)):
    cv2.imwrite(output_path + '/' + str(img_id) + '.jpg', \
                visualized[img_id])
```

BigDL が注目を集めている理由

BigDL は引き続き開発中ですが、初期の体験とフィードバックから見る将来は有望です。2016 年 12 月に最初にリリースされて以来、GitHub* で 2,400 を超えるスターを獲得し、Hadoop* や Apache Spark* ベースのクラスター上で多くの新しい解析およびディープラーニング・アプリケーションの開発が行われてきました。新しい Analytics Zoo プロジェクトは、Apache Spark* と BigDL 上でエンドツーエンド解析と AI プラットフォームを提供することで、BigDL アプリケーションの開発をさらに容易にします。

参考資料 (英語)

- Alex Heye, et al. "**Scalable Deep Learning with BigDL on the Urika-XC Software Suite**."
- Jason (Jinquan) Dai, et al. "**Leveraging Low Precision and Quantization for Deep Learning Using the Amazon EC2 C5 Instance and BigDL**."
- Jason (Jinquan) Dai, et al. "**Building Large-Scale Image Feature Extraction with BigDL at JD.com**."
- B. Zhou, et al. "**Places: A 10 Million Image Database for Scene Recognition**," IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI), 2017.

コンパイラーの最適化に関する詳細は、**最適化に関する注意事項**を参照してください。

Software

テクノロジーの未来を解読する

開発者向けの新しいナレッジハブ、Tech.Decoded へようこそ

現代のハードウェアを最大限に活用できるように、さまざな情報ライブラリーを公開しています。競争力を高め、市場投入期間を短縮しましょう。

- 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
- データサイエンス
- システムと IoT

エキスパートの知見を得る

テクノロジーの将来を予測する人々による、今日の技術展望と最もホットなトレンドについて視聴できます。

さらに深く掘り下げる

今日の最も重要な戦略、プラクティス、ツールに関するオンデマンドのセミナーからコードのパフォーマンスを最大限に引き出す方法が分かります。

概念のコード化

特定の開発ツールで主なプログラミング・タスクを実行する方法を説明した短いビデオや記事を利用できます。

Tech.Decoded を見る (英語) >

コンパイラーの最適化に関する詳細は、最適化に関する注意事項 (software.intel.com/en-us/articles/optimization-notice#opt-jp) を参照してください。
Intel、インテル、Intel ロゴは、アメリカ合衆国および / またはその他の国における Intel Corporation の商標です。
© Intel Corporation

WebAssembly が Web 上のコンピューティングの未来を握る理由

ブラウザーでの処理の歴史と新たな動向

Rich Winterton インテル コーポレーション ソフトウェア・エンジニア
Deepti Aggarwal 同ソフトウェア・エンジニア
Tuyet-Trang (Snow) Lam Piel 同ソフトウェア・エンジニア
Brittney Coons 同ソフトウェア・エンジニア・インターン
Nathan Johns 同ソフトウェア・エンジニア・インターン

1983 年に米国国防総省のプロジェクトである高等研究計画局ネットワーク (ARPANET)[1] は、TCP/IP ネットワーク・プロトコルを採用し、インターネットが誕生しました。当初、インターネットはデータとファイル共有に使用されることがほとんどでした。1992 年に商用利用が始まりましたが、それ以来インターネットの機能は拡張され、ユースケースは多岐にわたり、間違いなく世界で最も重要なテクノロジーになりました。

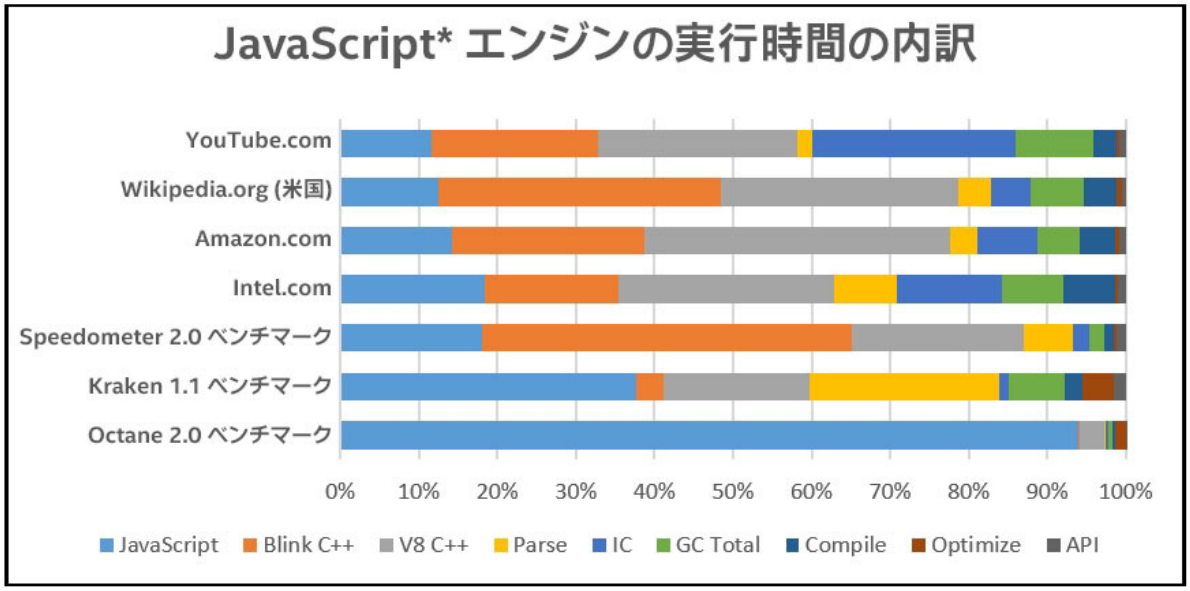
ブラウザーは、クライアントのコンピューターにコンテンツを配信するソフトウェア・アプリケーションです。当初は、単にワールド・ワイド・ウェブでデータにアクセスするためのアプリケーションにすぎませんでした。時間の経過とともに、ゲーム・アプリケーションやハードウェア・コーデックなどより複雑な Web アプリケーションで使用されるようになり、計算とパフォーマンスの要件が高まりました。

さまざまなブラウザーが市場シェアを競うようになり、Apple* の Sunspider とその後継である JetStream[2]、Mozilla* の Kraken[3]、Google* の Octane 1.0 とその後継である Octane 2.04 など、ブラウザーの速度を評価する合成ベンチマークが作成されました。これらのベンチマークは、ブラウザーの速度を比較するだけではなく、JavaScript* エンジンによって生成されるコードを評価し改善するために開発されました。多くの合成ブラウザー・ベンチマークは、JavaScript* のパフォーマンスを重視し、ハードウェアとソフトウェアの改善に伴い、JavaScript* の使用が増加しました。

その一方で、メディアとゲーム・アプリケーションのパフォーマンスを向上するため、Adobe* Flash* や Google* の Native Client (NaCl*) などのブラウザープラグインが登場しました。プラグインは Web 標準ではなかったため、多くのプラットフォームではビデオの高速化が困難でした。Adobe* Flash* Stage Video などの実装を、さまざまなプラットフォームのさまざまなブラウザーで使用できるようにすることは困難な作業でした。YouTube* とインテルのエンジニアは、ハードウェア・ベンダーがハードウェアおよびソフトウェア・ビデオ・デコードのパフォーマンス問題を特定して解決するのを支援するため、YouTube* の **Stats for Nerds** (英語) インターフェイスを作成しました。この頃、グラフィックス・プロセシング・ユニット (GPU) を別のプロセスに配置するサンドボックス・テクノロジーにより、ブラウザーベンダーは Web ページのレンダリングに GPU を使用できるようになりました。これにより、ほとんどのプラットフォームで安全なハードウェア合成とレンダリングが可能になり、不正な Web サイトがブラウザーやシステムをロックできないようになりました。

2015 年には、ブラウザーベンダーは、合成ベンチマークのパフォーマンスよりも、ブラウザーの実際の使用法に注目し始めました。**図 1** は、一般的な Web サイトとベンチマークの実行中に異なるステージで費やされた時間の割合を示しています。ほとんどの主要ブラウザーで同様の結果になることが予測されます。**図 1** から、Web サイト (YouTube.com、Wikipedia.org、Amazon.com、Intel.com) は、Kraken および Octane 合成ベンチマークと比較して、JavaScript* の JIT (Just-In-Time) コードで費やされた時間が非常に少ないことが分かります。これらのベンチマークには、ブラウザーの JavaScript* エンジンが、ベンチマーク・コードを利用してパフォーマンス結果を操作するという問題も生じました。2014 年に Apple* は、実際の Web サイトの機能を模倣したより現実的なベンチマーク、Speedometer をリリースしました。その類似性は**図 1** でも明らかです。

インテル® C++ コンパイラー
アプリケーションのパフォーマンスをスピードアップ
詳細



1 **インテル® Core™ i7-6770HQ プロセッサー 2.60GHz 上の Ubuntu* 16.04 で Google Chrome* を使用して実行した結果。パーセント表記は、各領域で費やされた時間の割合に対応します。**

JavaScript* コードのコンパイルは、多くの時間、メモリー、電力を消費します。Web アプリケーションのフレームワークとしてのブラウザー・コンポーネントとブラウザーの使用に対する需要の高まりを受けて、ベンダーはすぐにパフォーマンスを次のレベルに向上する作業が必要であることに気付きました。そして、典型的な Web サイトと将来の Web アプリケーション・モデル向けにブラウザーの設計の変更に取り掛かりました。Google* と Mozilla* は、これらの変更を早期に採用し、新しい Web テクノロジーに対するニーズの誕生と受け入れのきっかけとなる 2 つのテクノロジー、asm.js[5] と NativeClient[6] (NaCl*) をリリースしました。ブラウザーベンダーはすぐに、広く受け入れられる新しい Web テクノロジーの開発は困難であることに気付きました。この問題に協力して取り組むため、Apple*、Google*、Mozilla* を含むいくつかのブラウザーベンダーが集結して WebAssembly[7] (WASM) が誕生しました。この新しいテクノロジーは、コードを新しいタイプのバイトコードにコンパイルする代わりに、ネイティブコードやマネージドコードで通常使用される言語をコンパイルするように設計されました。WASM によって生成されるバイトコードは、汎用アプリケーションでネイティブに近いコンパイル・パフォーマンスを提供するように設計された可搬性の高いスタックマシンです。可搬性の高いインフラストラクチャーと優れたパフォーマンスを備えた WASM により、Web アプリケーションはネイティブ・アプリケーションのようになり、効率良く実行できるようになりました。また、インターネットに接続されていない状態でも実行できるようになりました。

プログレッシブ Web アプリケーション[8] (PWA) は、急速に勢いを増しています。PWA は、必要なリソースを最初にローカルで探して、利用可能な場合はそれを使用します。アプリケーションはインターネットに接続すると、ローカルリソースが最新の状態かどうかチェックします。最新の状態の場合、アプリケーションはオフラインで使用する準備をします。新しいまたは更新されたリソースがある場合、アプリケーションはローカル・データベースを更新します。この機能は、2017 年 11 月に W3C がサービスワーカー向けのドラフト仕様[9] を公開するまで Web 標準ではありませんでした。サービスワーカーは、ネットワークからデータを取得する前に、PWA がキャッシュされたアセットを使用または更新できるようにします。これは、「offline first」と呼ばれる手法です。

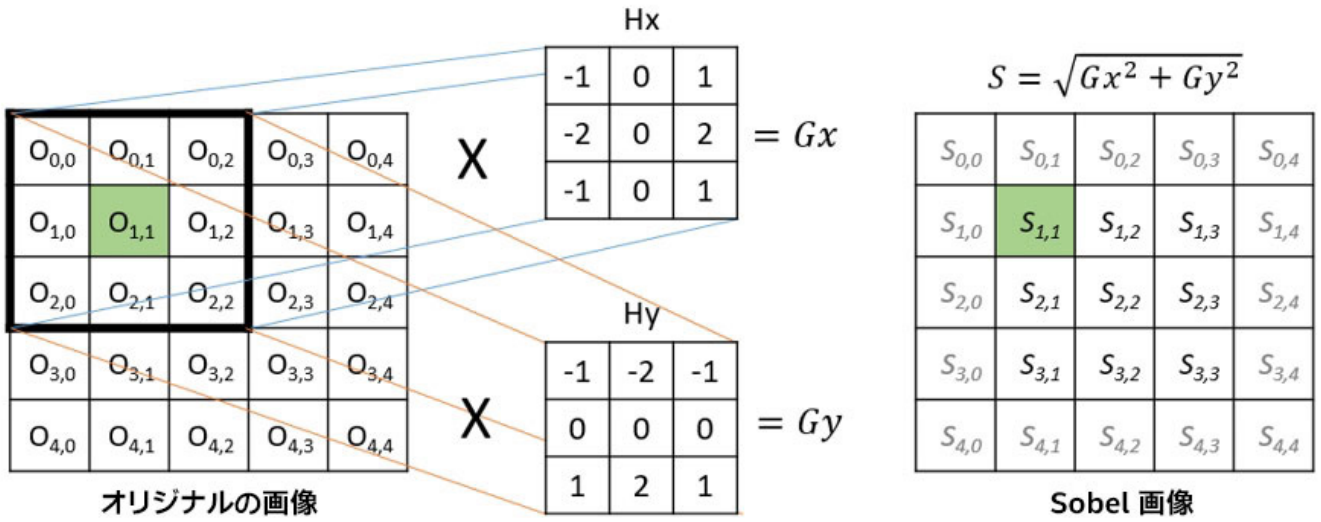
PWA は、機能的にはほとんどのネイティブ・アプリケーションと似ていますが、小さく、ユーザーによる明示的なインストールや更新をほとんど必要としません。最もよく使用されている PWA には、Gmail*、Twitter*、Google Maps* などがあります。通常、インストールの 24 時間後にインストールされたままのアプリケーションは 25% で[10]、90 日後には 5% 未満になります。PWA はインストールや更新をほとんど必要としないため、リテンションが大幅に改善され、ユーザーのアクティビティーやエンゲージメントからその価値を測ることができます。例えば、Twitter* はモバイル PWA になることで、コンテンツ閲覧数が 65% 増加し、ユーザー・エンゲージメント (ツイートの送信) が 75% 増加し、直帰率 (最初のページを訪問した後に、アプリケーションから離脱した割合) は 3% 未満になったと報告しています。[11] 別の例として、中国のオンライン小売サービス AliExpress* は、コンバージョン率 (訪問数に対する購入数の比率) を 2 倍以上向上しました。[12] このような改善は大幅な収益の増加につながります。その結果、多くのビジネスが PWA へ移行しており、Forbes* は「PWA はネイティブ・モバイル・アプリケーションに取って代わる」と予測しています。[13]

WASM は、計算集約型タスクのパフォーマンスを JavaScript* と比較して最大 3 倍向上できる、現在および将来の Web アプリケーションのパフォーマンス要求に対応する鍵と言えます。このパフォーマンスの向上は PWA に大きな利点をもたらします。

Web コンピューティングのブラウザーへの移行

Web コンピューティングをブラウザーへ移行する方法を見るため、**図 2** に示す静止画のエッジ検出の例について考えてみます。この例は、Sobel フィルター・アルゴリズムを使用する単純なエッジ検出[14] であり、ネイティブコード実装をベースラインとした場合に JavaScript* と WASM の間に大幅なパフォーマンスの違いがあることを示します。このアルゴリズムは、入力画像 (左) を受け取り、相対エッジを計算して、エッジの強度に応じて黒から白のスペクトラム (右) で表示します。画像は、グレースケールとしてインポートされるか、グレースケールに変換されます。アルゴリズムは、オリジナルの画像をピクセルごとに反復して、各ピクセルとその周辺のピクセルを比較し、Sobel 画像用の新しいピクセルを計算します (**図 3**)。

2 **Sobel エッジ検出を適用する前と後の画像**



3 **Sobel エッジ検出アルゴリズム**

図 3 に示すように、オリジナルの画像データと 2 つのフィルターとの行列乗算を行います。Hx フィルターは、x 方向 (Gx) のピクセルデータの相対的な変化を計算し、Hy フィルターは y 方向 (Gy) のピクセルデータの相対的な変化を計算します。オリジナルの画像で検出されたエッジを正確に表現するには、両方のフィルターが必要です (**図 4**)。

Hx filter　　Hy filter

4 **Sobel フィルター Hx または Hy のいずれかを使用して作成された画像**

Sobel エッジ検出アルゴリズムの C++ と JavaScript* 実装を**図 5** と **6** にそれぞれ示します。入力データの読み取りは、ローカルファイルの読み取りから HTML キャンバスのデータの取得に変更する必要がありました。それ以外には、ほとんど変更は必要ありませんでした。JavaScript* コードは、入力画像を受け取り、RGBA 画像データに変換して、Sobel フィルター関数に渡します。そして、キャンバスに表示するため、データをブラウザーに戻します。

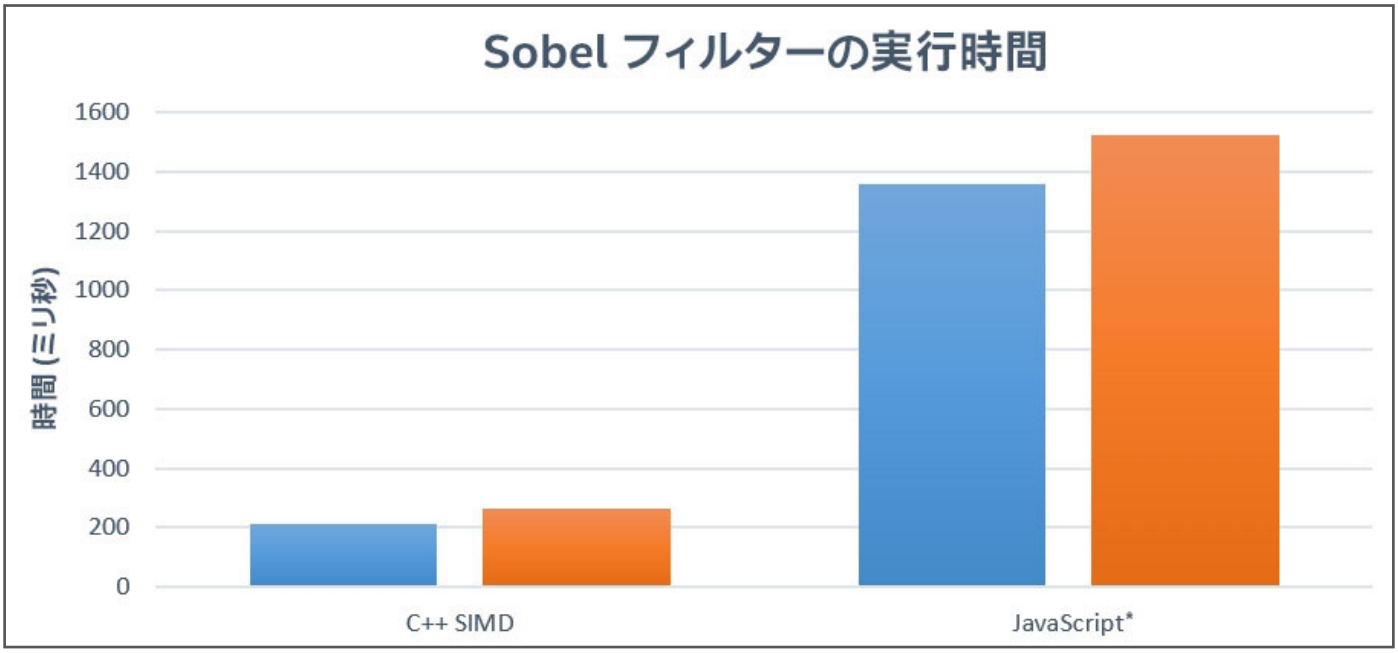
C++ と JavaScript* 実装はほぼ同じです。そのため、合理的なパフォーマンスの比較が可能です (**図 7**)。ネイティブ C++ 実装は、JavaScript* 実装よりも約 6 倍高速です。つまり、エッジ検出のような複雑な計算では高い計算能力が求められることは明らかです。当然のことながら、高い計算能力によりさらに多くのアプリケーションで PWA テクノロジーが実行可能になります。ネイティブにコンパイルした C++、JavaScript* 生成コード、WASM のパフォーマンスを比較するため、Emscripten*[16] を使用して Sobel C++ ソースコードを WASM にコンパイルして、ブラウザー内で実行しました。**図 8** に示すように、WASM はネイティブバイナリーと JavaScript* の間のパフォーマンス・ギャップを埋めます。

```
1   float magnitudeValue(unsigned char **imageData, int row, int col, int imageHeight, int imageWidth) {
2       //accumulation1:
3       if ((row != 0) && (col != 0) && (row != imageHeight - 1) && (col != imageWidth - 1)) { // Center of data
4           accumulation1_1 = (float)(Hx[0][0] * imageData[row - 1][col - 1] + Hx[0][1] * imageData[row - 1][col] +
5               Hx[0][2] * imageData[row - 1][col + 1]);
6
7           accumulation1_2 = (float)(Hx[1][0] * imageData[row][col - 1] + Hx[1][1] * imageData[row][col] +
8               Hx[1][2] * imageData[row][col + 1]);
9
10          accumulation1_3 = (float)(Hx[2][0] * imageData[row + 1][col - 1] + Hx[2][1] * imageData[row + 1][col] +
11              Hx[2][2] * imageData[row + 1][col + 1]);
12      }
13
14      // Repeat with Hy to calculate accumulations2
15      accumulation1 = accumulation1_1 + accumulation1_2 + accumulation1_3;
16      magnitude = accumulation1 * accumulation1 + accumulation2 * accumulation2;
17      return magnitude;
18  }
19  unsigned char** sobelFilter(unsigned char** imageData, int imageHeight, int imageWidth) {
20      unsigned char** outputData = new unsigned char*[imageHeight];
21      for (int row = 0; row < imageHeight; row++) {
22          outputData[row] = new unsigned char[imageWidth];
23          for (int col = 0; col < imageWidth; col++) {
24              mag = sqrt(magnitudeValue(imageData, row, col, imageHeight, imageWidth));
25              mag = (mag > 255) ? 255 : mag;
26              outputData[row][col] = mag;
27          }
28      }
29      return outputData;
30  }
```

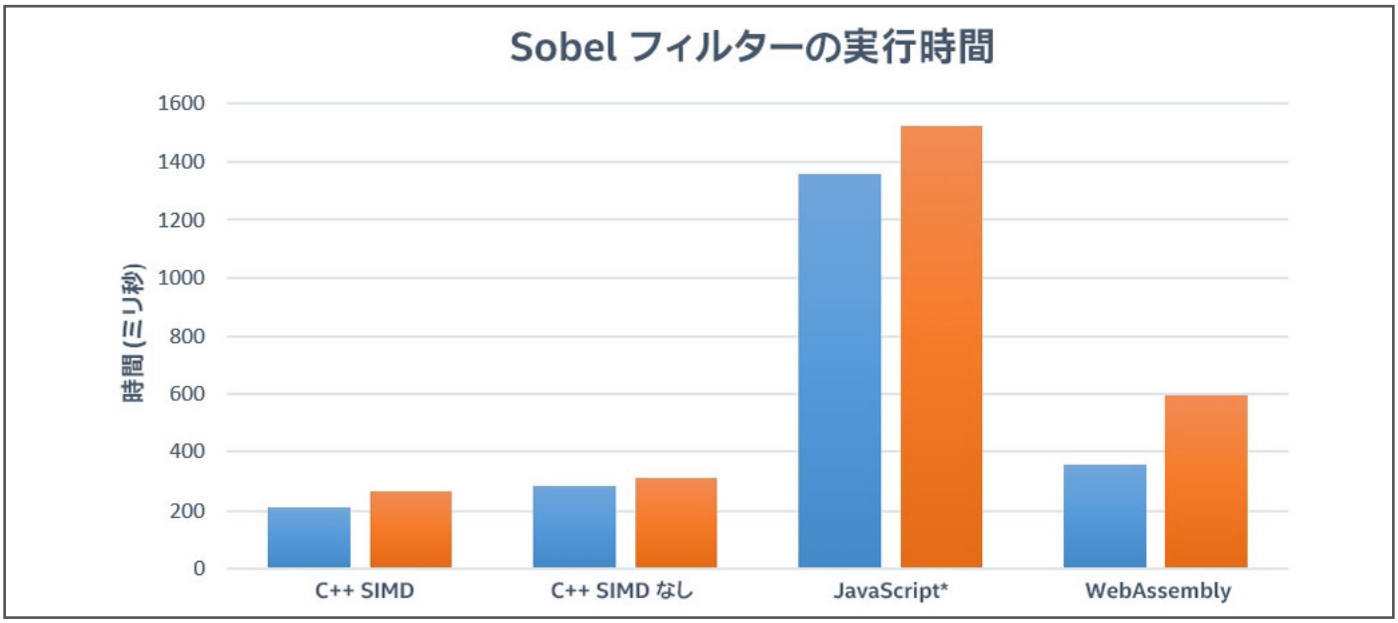
5 **Sobel フィルター・アルゴリズムの C++ コード (完全なソースコードは GitHub* から入手可能[15])**

```javascript
const magnitudeValue = (imageData, row, col, imageHeight, imageWidth) => {
    if ((row !== 0) && (col !== 0) && (row !== imageHeight - 1) && (col !== imageWidth - 1)) { // Center of data
        accumulation1_1 = Hx[0][0] * imageData[row - 1][col - 1] + Hx[0][1] * imageData[row - 1][col] +
            Hx[0][2] * imageData[row - 1][col + 1];

        accumulation1_2 = Hx[1][0] * imageData[row][col - 1] + Hx[1][1] * imageData[row][col] +
            Hx[1][2] * imageData[row][col + 1];

        accumulation1_3 = Hx[2][0] * imageData[row + 1][col - 1] + Hx[2][1] * imageData[row + 1][col] +
            Hx[2][2] * imageData[row + 1][col + 1];
    }
    accumulation1 = accumulation1_1 + accumulation1_2 + accumulation1_3;

    // Repeat with Hy to calculate accumulation2

    let magnitude = accumulation1 * accumulation1 + accumulation2 * accumulation2;
    return magnitude;
};

const sobelFilter = (width, height) => {
    for (let row = 0; row < height; row++) {
        sobelImageData[row] = new Array(width);
        for (let col = 0; col < width; col++) {
            let mag = magnitudeValue(grayscaleData, row, col, height, width);
            mag = Math.sqrt(mag);
            mag = (mag > 255) ? 255 : mag;
            sobelImageData[row][col] = mag;
        }
    }
};
```

6 Sobel フィルター・アルゴリズムの JavaScript* コード (完全なソースコードは GitHub* から入手可能[15])



7 4,160 x 3,160 ピクセルの画像での Sobel フィルターのパフォーマンス - インテル® Core™ i7-6770HQ プロセッサー 2.60GHz 上の Ubuntu* 16.04 (青色のバー) とインテル® Core™ i5-7300U プロセッサー 2.60GHz 上の Windows® 10 (オレンジ色のバー)。JavaScript* では、時間の計測に performance.now() を使用しました。C++ では、時間の計測に高精度クロックを使用しました。JavaScript* のテストは Google Chrome* を使用して実行しました。



8 **4,160 x 3,160 ピクセルの画像での Sobel フィルターのパフォーマンス - インテル® Core™ i7-6770HQ プロセッサー 2.60GHz 上の Ubuntu* 16.04 (青色のバー) とインテル® Core™ i5-7300U プロセッサー 2.60GHz 上の Windows® 10 (オレンジ色のバー)。**

将来の展望

将来についてはさまざまな推測がありますが、トレンドは明確になりつつあります。

- **PWA** が多くのモバイルおよびネイティブ・アプリケーションに取って代わるでしょう。
- **Web テクノロジー**はデータを送受信する最も単純で最良の手段です。
- **マシンラーニングと人工知能 (AI)** はアプリケーションをカスタマイズ可能にし、ユーザーへの関連付けを高めるだけでなく、PWA 実装自体を改善するため、家庭や補助デバイス / アプリケーションで使用されます。

これにより、クライアントとエッジ・コンピューティングの重要性が大幅に増します。

WASM コードの並列および JIT コンパイルなど、WebAssembly のいくつかの短期的な拡張がこの必要性を高めています。この手法は、マルチコア・プラットフォームを利用し、クライアントによってコンパイルされるアプリケーションの初回実行時または更新時の実行時間を大幅に向上します。メモリー管理は、JavaScript*/WASM メモリー管理ソフトウェア内のガベージ・コレクションを並列実装に移行することで改善できます。アプリケーションのネイティブ実装は、コードが非リアルタイムにコンパイルされ、自動ベクトル化や手作業でカスタマイズされた SIMD の利点を簡単に得られるため、複雑な計算が必要な場合、依然として最良のパフォーマンスを提供します。しかし、近い将来 WASM が LLVM WASM フレームワークで C/C++ コードをコンパイルおよびリンクするだけで自動ベクトル化できる機能を実装したら、この利点は最小限になるでしょう。

コンパイラーの最適化に関する詳細は、**最適化に関する注意事項**を参照してください。

今後の需要への対応

Web ブラウザーは、インターネット革命で大きな役割を果たしたアプリケーションです。最初は、クライアントにコンテンツを提供するだけのアプリケーションでした。しかし、パフォーマンスへの要求が高まり、新しいユースケースの誕生により、ブラウザーは大きく変わりました。処理能力とパフォーマンスの問題に対応するため新しいベンチマークと言語が登場しましたが、現在と将来の Web アプリケーションの要求を満たすためには、さらなるパフォーマンスが必要であることがすぐに明らかになりました。

WASM は、このパフォーマンス要求に対応する新しいテクノロジーです。通常ネイティブコードやマネージドコードで使用される言語をコンパイルするように設計されています。可搬性の高いバイトコードを生成し、汎用アプリケーションでネイティブに近いコンパイル・パフォーマンスを提供します。

WASM の利点を示すため、C++、JavaScript*、WASM で Sobel フィルター・アルゴリズムを実装しました。予想どおり、SIMD を利用しない場合でも、JavaScript* と比較して 3 倍から 4 倍のパフォーマンスを達成しました。SIMD 自動ベクトル化などの新しい機能が追加されれば、WASM は現在および将来の PWA のパフォーマンス要求を満たすことができる主要テクノロジーになるでしょう。

参考文献 (英語)

1. Featherly, Kevin. "**ARPANET United States Defense Program**," Encyclopaedia Britannica, 28 November 2016.
2. **JetStream ベンチマーク**
3. **Kraken ベンチマーク**
4. **Octane ベンチマーク**
5. **asm.js**
6. **Native Client (NaCl)**
7. **WebAssembly**
8. **プログレッシブ Web アプリケーション (PWA)**
9. **Service Workers の W3C 仕様**
10. Grennan, Todd. "**App User Retention Spring 2016 Report**," Braze Magazine,19 May 2016.
11. "**Twitter Lite PWA Significantly Increases Engagement and Reduces Data Usage**," Google, Google Developers, 17 May 2017.
12. "**AliExpress**," Google, Google Developers, 9 February 2017.
13. Gazdecki, Andrew. "**Why Progressive Web Apps Will Replace Native Mobile Apps**," Forbes, 9 March 2018.
14. Sobel Filter: Jain, Anil K. Fundamentals of Digital Image Processing. Prentice-Hall, 1989.
15. **サンプル・ソースコード**
16. **Emscripten* の設定**

コンパイラーの最適化に関する詳細は、**最適化に関する注意事項**を参照してください。

intel Software

より高速な Python* によるデータ処理

インテル® Distribution for Python* で
アプリケーションをスーパーチャージ

無料のダウンロード (英語) >

コンパイラーの最適化に関する詳細は、最適化に関する注意事項 (software.intel.com/en-us/articles/optimization-notice#opt-jp) を参照してください。
Intel、インテル、Intel ロゴは、アメリカ合衆国および / またはその他の国における Intel Corporation の商標です。
* その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。
© Intel Corporation

コードの現代化を実践：スレッド化、メモリー、ベクトル化の最適化

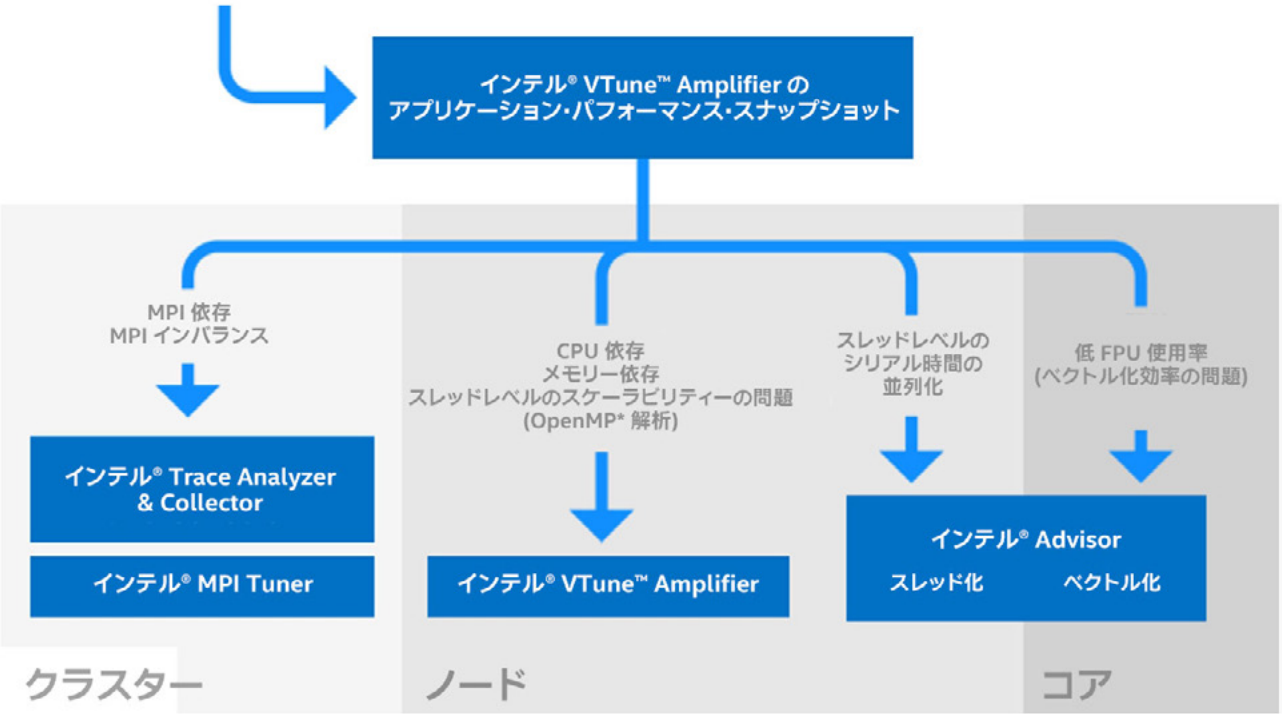
最新のインテル® ソフトウェア開発ツールを利用してハードウェアを効率良く使用する

Dmitry Prohorov インテル コーポレーション インテル® VTune™ Amplifier HPC リード
Cedric Andreolli 同テクニカル・コンサルティング・エンジニア
Philippe Thierry 同主席エンジニア

ハイパフォーマンス・コンピューティング (HPC) システムは、ハードウェア・コンポーネントの複雑な組み合わせであると言えます。ハードウェア・ベンダーは、計算集約型アプリケーションのパフォーマンスを向上するため、スピードアップ、レイテンシーの軽減、コアの追加、ベクトル幅の拡張に常に取り組んでいます。しかし、単により優れたコンポーネントを備えたシステムでアプリケーションを実行するだけでは、それらのコンポーネントが効率良く使用される保証はありません。期待するパフォーマンス向上を達成するため、コード変更 (コードを対応させること) が必要になることがあります。

幸いにも、**インテル® Parallel Studio XE** には、インテル® ハードウェア上でコードのチューニング作業を支援するツールが含まれています。どのツールから使用すべきか迷った場合は、**インテル® VTune™ Amplifier の**

アプリケーション・パフォーマンス・スナップショット (APS) (英語) を使用することで、アプリケーションのパフォーマンス特性、特にパフォーマンスを制限しているボトルネックのサマリーを素早く取得できます。アプリケーション・パフォーマンス・スナップショットは、クラスター、ノード、コアレベルのパフォーマンスのチューニングに適切なインテル® Parallel Studio XE ツールを選択するのにも役立ちます (**図 1**)。



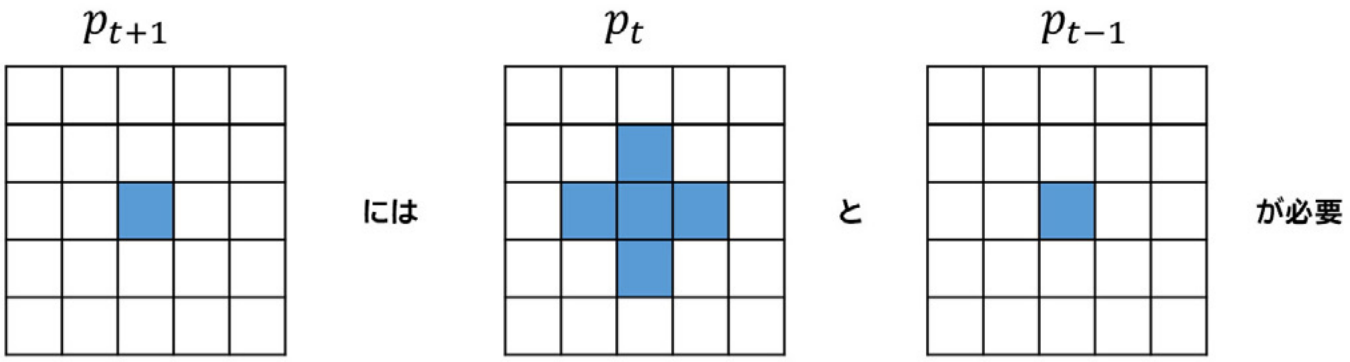
1 **APS と専用のパフォーマンス・ツール**

ISO3DFD: 波動伝播カーネルとパフォーマンス測定

サンプルコードを使用して、ボトルネックの発見とパフォーマンスの向上にツールがどのように役立つかを見てみましょう。ISO3DFD は、等方性音波方程式を実装します。

$$\frac{\partial^2 p}{\partial t^2} = c^2 \nabla^2 p$$

ここで、∇^2 はラプラス演算子、p は圧力場、c は速度場です。有限差分は、p_t と p_{t-1} の関数として p_{t+1} を表現するために使用できます。有限差分伝播カーネルは、ステンシルパターンを利用して実装できます。3 次元では、ステンシルは圧力場を横切って移動する 3D クロスのように見えます。つまり、$p_{t+1}[x,y,z]$ を更新するには、$p_t[x,y,z]$ のすべての近傍に 3D でアクセスする必要があります。**図 2** は、2D のステンシルパターンです。



2 **2D ステンシルパターン**

実際のアプリケーションでは、物理学者は通常コーナーケース用に特定の伝播手法を実装します。これらのコーナーケースは、境界条件と呼ばれます。ここではコードを簡潔にするため、境界に波を伝播しません。**図 3** の単純な実装は、ステンシルの単一の反復を計算します。

```
1  void iso_3dfd_it(float *ptr_next,  float *ptr_prev,  float *ptr_vel,   float *coeff, const int n1, const int n2, const int n3,
2  const int num_threads, const int n1_Tblock, const int n2_Tblock, const int n3_Tblock){
3  int dimn1n2 = n1*n2;
4  for(int ix=0; ix<n1; ix++) {
5    for(int iy=0; iy<n2; iy++) {
6      for(int iz=0; iz<n3; iz++) {
7        if( ix>=HALF_LENGTH && ix<(n1-HALF_LENGTH) &&
8            iy>=HALF_LENGTH && iy<(n2-HALF_LENGTH) &&
9            iz>=HALF_LENGTH && iz<(n3-HALF_LENGTH) ) {
10           int offset = iz*dimn1n2 + iy*n1 + ix;
11           float value = 0.0;
12           value += ptr_prev[offset]*coeff[0];
13           for(int ir=1; ir<=HALF_LENGTH; ir++) {
14              value += coeff[ir] * (ptr_prev[offset + ir] + ptr_prev[offset - ir]);// horizontal
15              value += coeff[ir] * (ptr_prev[offset + ir*n1] + ptr_prev[offset - ir*n1]);// vertical
16              value += coeff[ir] * (ptr_prev[offset + ir*dimn1n2] + ptr_prev[offset - ir*dimn1n2]); // in front / behind
17           }
18           ptr_next[offset] = 2.0f* ptr_prev[offset] - ptr_next[offset] + value*ptr_vel[offset];
19 }}}}}
```

3 **ステンシルの単一の反復の計算**

インテル® VTune™ Amplifier
現代のプロセッサーのパフォーマンス解析
無料評価版のダウンロード

この記事では単一ノードのパフォーマンスに注目するため、サンプルコードは MPI を使用しておらず、クラスターレベルの並列処理を実装しません。

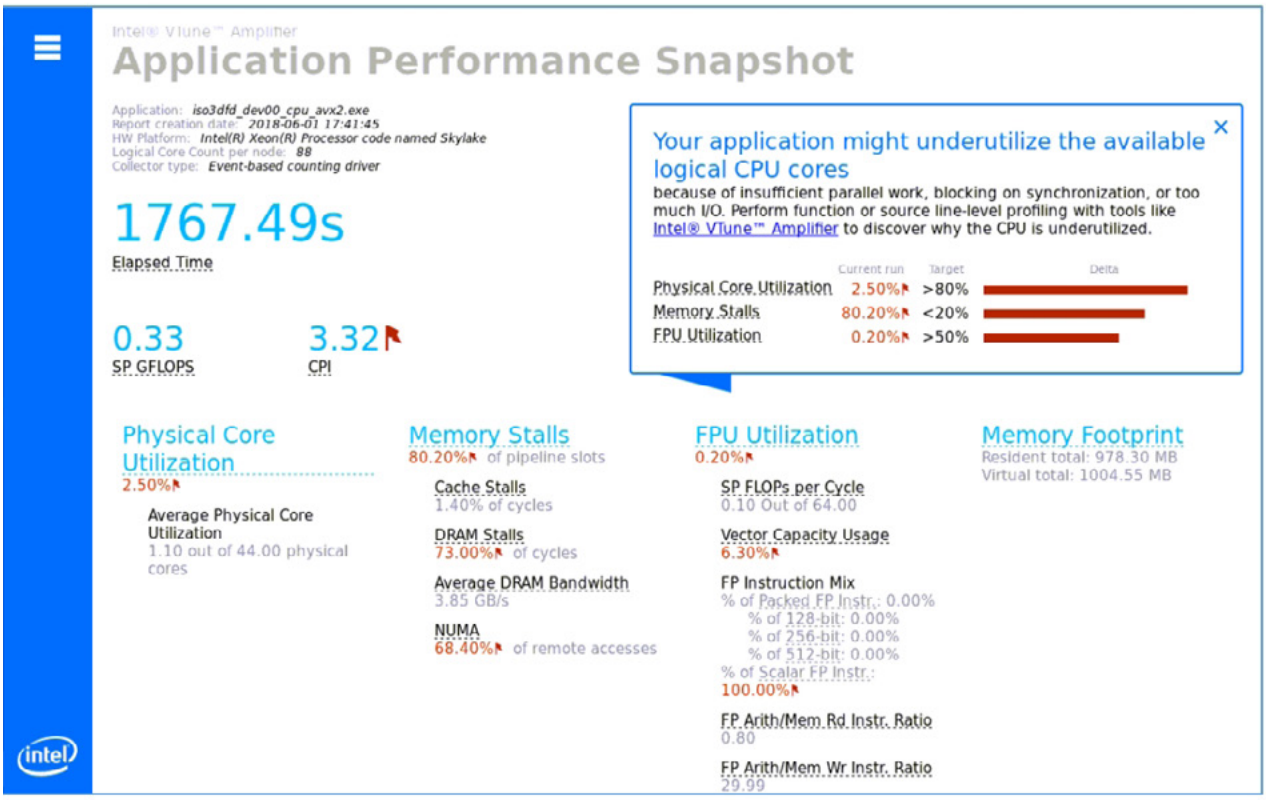
ベースライン・パフォーマンスを確立するため、デュアルソケットの**インテル® Xeon® Gold 6152 プロセッサー** (2.10GHz、22 コア / ソケット) を搭載した Red Hat* Enterprise Linux* Server 7.4 システム上で、512 x 400 x 400 グリッドで初期実装を実行してベンチマークを測定しました。次のコマンドを使用して、インテル® Parallel Studio XE 環境を設定します。

```
> source <Parallel_Studio_install_dir>/psxevars.sh
```

そして、インテル® VTune™ Amplifier のアプリケーション・パフォーマンス・スナップショットを使用して、ISO3DFD 初期実装のチューニングの可能性を素早く見つけます。次のコマンドで APS を起動します。

```
> aps ./Iso3DFD
```

図 4 は、Web ブラウザーに表示される APS レポートです。アプリケーションの初期のシングルスレッド・バージョンは、システムの 44 物理コアのうち 1 つのみを使用します。これは、APS でパフォーマンス向上の大きな可能性と認識されます。コードを並列化してみましょう。



4 **オリジナルのシングルスレッド・コードに対する APS レポート**

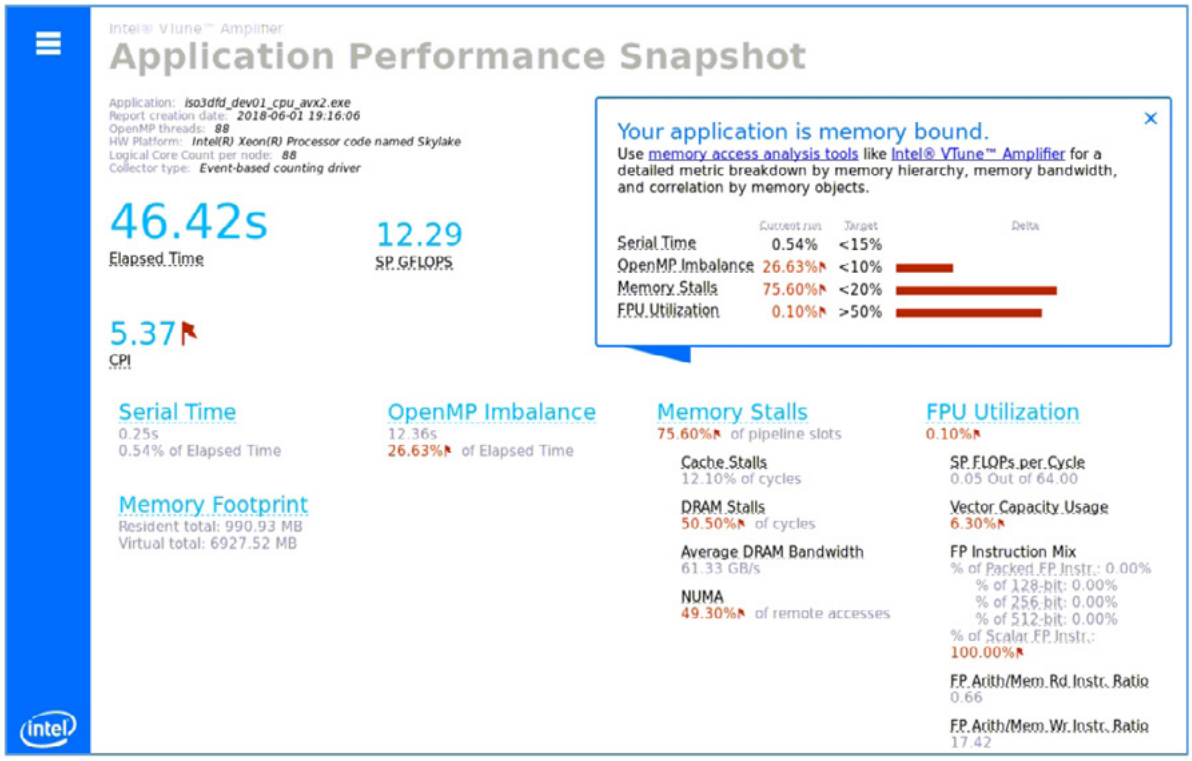
最適化ステップ 1: OpenMP* を使用してスレッドレベルの並列処理を追加する

ループに依存関係がないため、反復あたりの作業量が最大である最外ループに OpenMP* の "parallel for" プラグマを追加して並列化を行います (**図 5**)。

```
1  void iso_3dfd_it(float *ptr_next,  float *ptr_prev,  float *ptr_vel,   float *coeff, const int n1, const int n2, const int n3,
2  const int num_threads, const int n1_Tblock, const int n2_Tblock, const int n3_Tblock){
3    int dimn1n2 = n1*n2;
4    #pragma omp parallel for default(shared)
5  for(int ix=0; ix<n1; ix++) {
6    for(int iy=0; iy<n2; iy++) {
7      for(int iz=0; iz<n3; iz++) {
8        if( ix>=HALF_LENGTH && ix<(n1-HALF_LENGTH) &&
9            iy>=HALF_LENGTH && iy<(n2-HALF_LENGTH) &&
10           iz>=HALF_LENGTH && iz<(n3-HALF_LENGTH) ) {
11           int offset = iz*dimn1n2 + iy*n1 + ix;
```

5 **最外ループに OpenMP* の "parallel for" プラグマを追加する並列化**

APS でクイックチェックを行うと、経過時間 (Elapsed Time) が大幅に短縮されたことが分かります (**図 6**)。OpenMP* のロード・インバランスが報告されていますが、約 38 倍のスピードアップは良いスタートと言えます。ここで優先度の高いパフォーマンスの問題は、アプリケーションがメモリー依存であることです。このボトルネックを排除するため、メモリーアクセスを改善する必要があります。



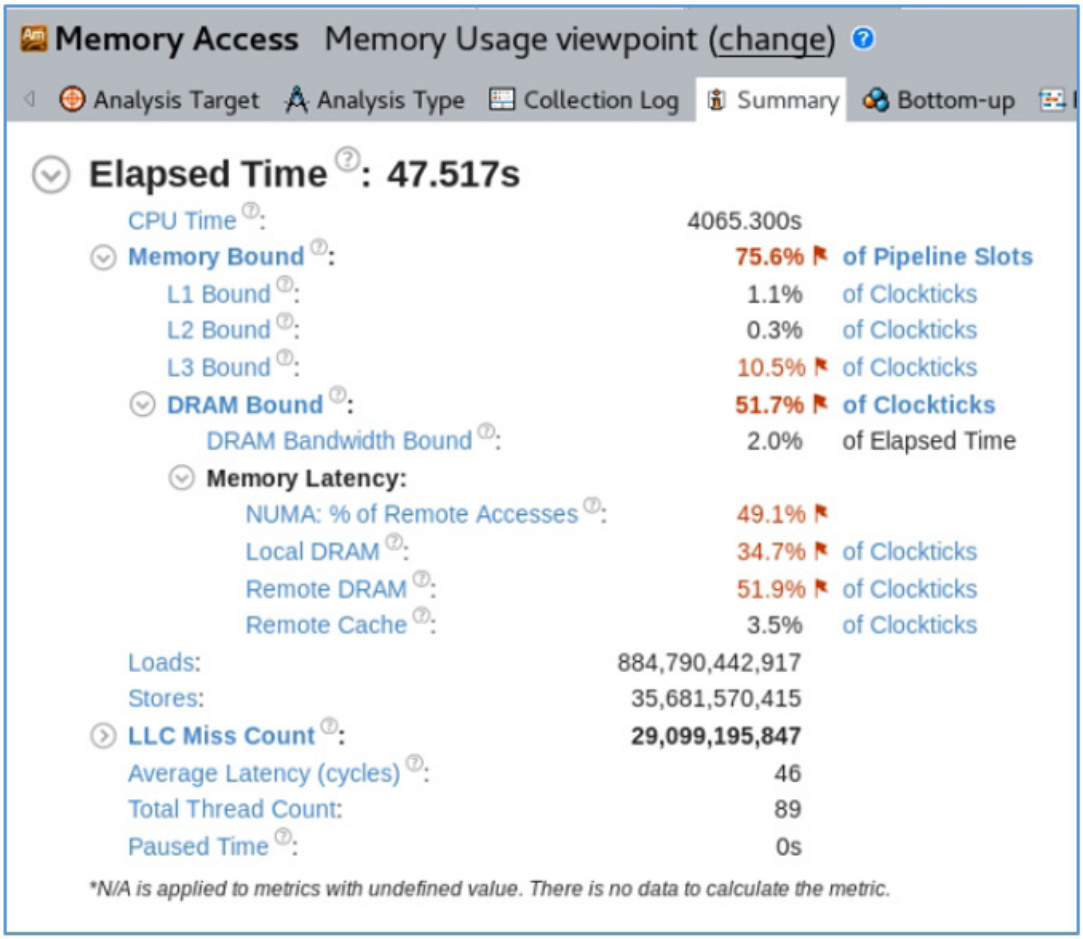
6 **OpenMP* による並列化後の APS レポート**

最適化ステップ 2: ループ交換によりメモリーアクセスを改善する

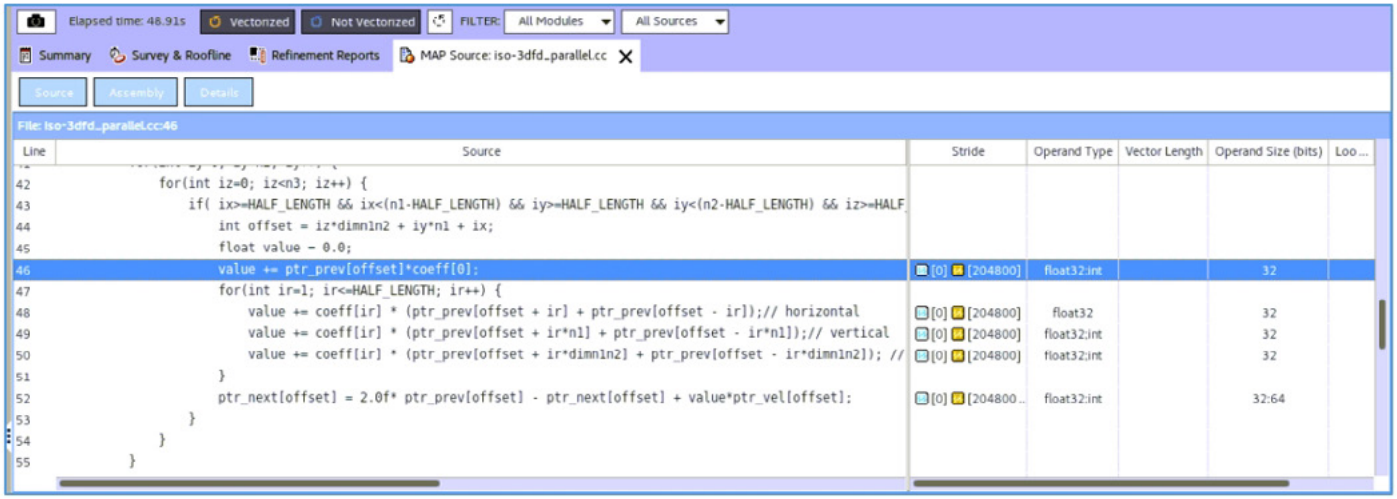
最適ではないメモリーアクセスによるメモリーストールが、アプリケーションの主要ボトルネックであるため、APS が推奨するインテル® VTune™ Amplifier のメモリーアクセス解析を実行します (**図 6**)。

```
> amplxe-cl -collect memory-access ./Iso3DFD
```

解析結果を**図 7** に示します。アプリケーションは DRAM 依存ですが、メモリー帯域幅の使用量は上限を大きく下回っています (つまり、DRAM 帯域幅依存の値が小さい)。これは、長いストライドでメモリーにアクセスするアプリケーションにおいて典型的な結果です。**インテル® Advisor** のメモリー・アクセス・パターン解析は、問題の原因であるコード行の検出に役立ちます (**図 8**)。



7 **インテル® VTune™ Amplifier のメモリーアクセス解析のサマリー**



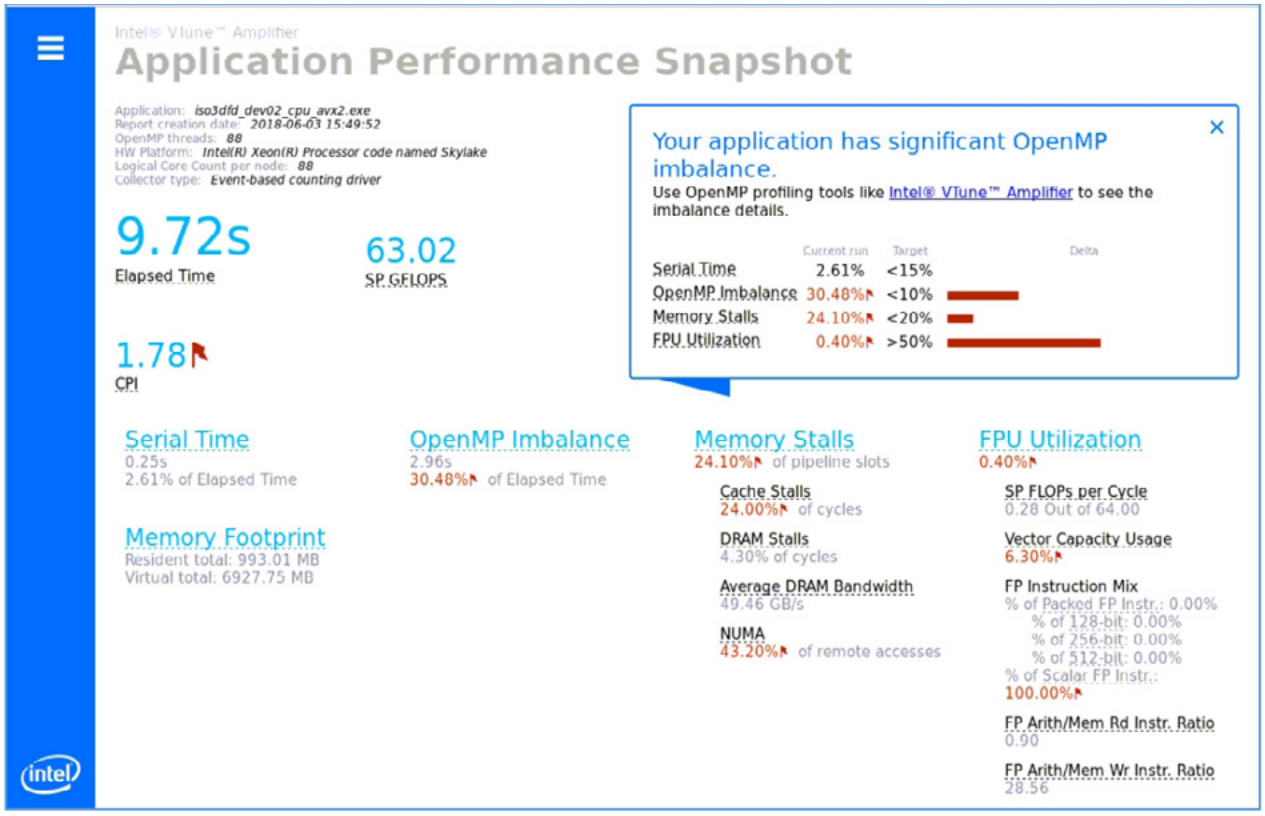
8 **インテル® Advisor のメモリー・アクセス・パターン解析**

204,800 ストライドは、最初の 2 つの次元のサイズです (512 x 400)。つまり、最内ループの配列要素による反復でプロセッサー・キャッシュが効率良く使用されていません。この問題は、ループの順序を変更することで解決できます (**図 9**)。

```
4  #pragma omp parallel for default(shared)
5  for(int iz=0; iz<n3; iz++) {
6    for(int iy=0; iy<n2; iy++) {
7      for(int ix=0; ix<n1; ix++) {
8        if( ix>=HALF_LENGTH && ix<(n1-HALF_LENGTH) &&
9            iy>=HALF_LENGTH && iy<(n2-HALF_LENGTH) &&
```

9 **ループの順序の変更**

この変更により、最内ループにおける配列要素へのメモリーアクセスが連続し、キャッシュの再利用が向上します。再コンパイルして、再度 APS でパフォーマンス・スナップショットを取得します (**図 10**)。レポートから、メモリーアクセスの最適化により約 4.7 倍のスピードアップを達成できたことが分かります。メモリーストールの割合が大幅に減少し、アプリケーションは DRAM 依存ではなくなりました。しかし、**図 6** と**図 10** で報告された OpenMP* のインバランスが主要なパフォーマンスの問題になっています。



10 ループ交換後の APS レポート

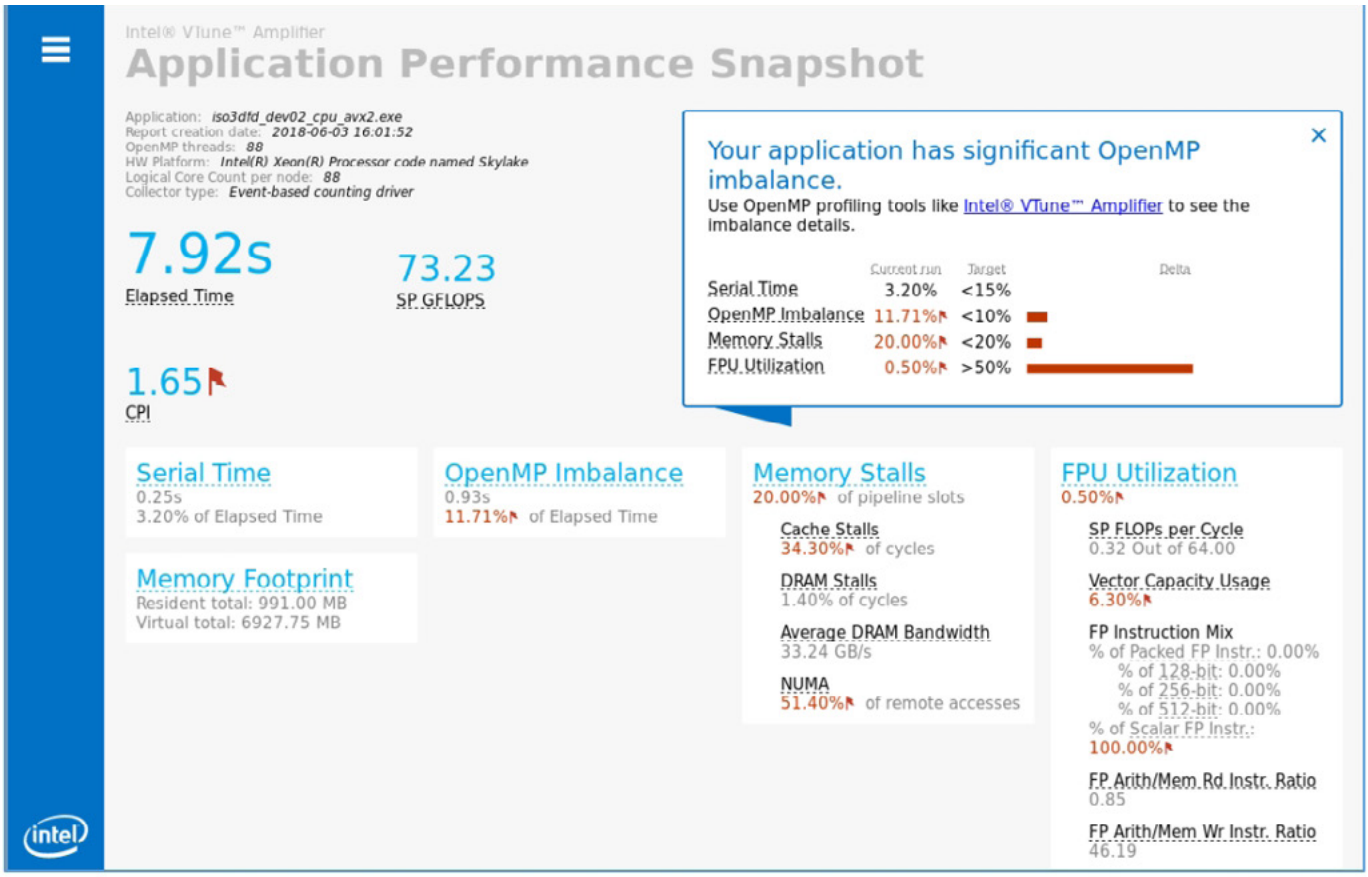
最適化ステップ 3: OpenMP* の動的スケジューリングによりロードバランスを改善する

まだメモリーストールが残っていますが、APS は経過時間の 30% を占める OpenMP* のインバランスに注目しています。OpenMP* は、デフォルトで静的スケジューリングを使用します。これは一般に有効で低オーバーヘッドのスケジューリング・プロトコルです。しかし、場合によってはロード・インバランスを引き起こすことがあるため、代わりに動的スケジューリングを試してみましょう (**図 11**)。

```
2  const int num_threads, const int n1_Tblock, const int n2_Tb
3  int dimn1n2 = n1*n2;
4  #pragma omp parallel schedule(dynamic) for default(shared)
5  for(int iz=0; iz<n3; iz++) {
6    for(int iy=0; iy<n2; iy++) {
7      for(int ix=0; ix<n1; ix++) {
8        if( ix>=HALF_LENGTH && ix<(n1-HALF_LENGTH) &&
```

11 OpenMP* の動的スケジューリング

これにより、ロード・インバランスが大幅に改善されます (**図 12**)。



12 **動的スケジューリング適用後の APS レポート**

最適化ステップ 4: キャッシュ・ブロッキングによりメモリーアクセスを改善する

キャッシュストールの割合がまだ大きいため (**図 12**)、キャッシュ・ブロッキングと呼ばれる最適化を実装して、キャッシュデータの再利用の改善を試みます (**図 13**)。

```
        int dimn1n2 = n1*n2;
        int n3End = n3 - HALF_LENGTH;
        int n2End = n2 - HALF_LENGTH;
        int n1End = n1 - HALF_LENGTH;
#pragma omp parallel for schedule(dynamic) collapse(3) default(shared)
        for(int bz=HALF_LENGTH; bz<n3End; bz+=n3_Tblock){
                for(int by=HALF_LENGTH; by<n2End; by+=n2_Tblock){
                        for(int bx=HALF_LENGTH; bx<n1End; bx+=n1_Tblock){
                                int izEnd = MIN(bz+n3_Tblock, n3End);
                                int iyEnd = MIN(by+n2_Tblock, n2End);
                                int ixEnd = MIN(n1_Tblock, n1End-bx);
                                int ix;
                                for(int iz=bz; iz<izEnd; iz++) {
                                        for(int iy=by; iy<iyEnd; iy++) {
                                                float* ptr_next = ptr_next_base + iz*dimn1n2 + iy*n1 + bx;
                                                float* ptr_prev = ptr_prev_base + iz*dimn1n2 + iy*n1 + bx;
                                                float* ptr_vel = ptr_vel_base + iz*dimn1n2 + iy*n1 + bx;
                                                for(int ix=0; ix<ixEnd; ix++) {
                                                        float value = 0.0;
                                                        value += ptr_prev[ix]*coeff[0];
                                                        for(int ir=1; ir<=HALF_LENGTH; ir++) {
                                                                value += coeff[ir] * (ptr_prev[ix + ir] + ptr_prev[ix - ir]);
                                                                value += coeff[ir] * (ptr_prev[ix + ir*n1] + ptr_prev[ix - ir*n1]);
                                                                value += coeff[ir] * (ptr_prev[ix + ir*dimn1n2] + ptr_prev[ix - ir*dimn1n2]);
                                                        }
                                                        ptr_next[ix] = 2.0f* ptr_prev[ix] - ptr_next[ix] + value*ptr_vel[ix];
                                                }
                                        }
                                }
                        }
                }
        }
```

13 **キャッシュ・ブロッキング**

OpenMP* スレッドに並列処理を割り当てるため、**`omp parallel for`** プラグマに **`collapse(3)`** 節を追加して、ブロッキング対象の 3 つのループが 1 つの大きな反復空間にコラプスされるようにします。新しい APS レポートでは、キャッシュ・ブロッキングによりパフォーマンスが 1.3 倍向上し、メモリーストールが減少し、OpenMP* のロードバランスが改善されています (**図 14**)。

BLOG HIGHLIGHTS

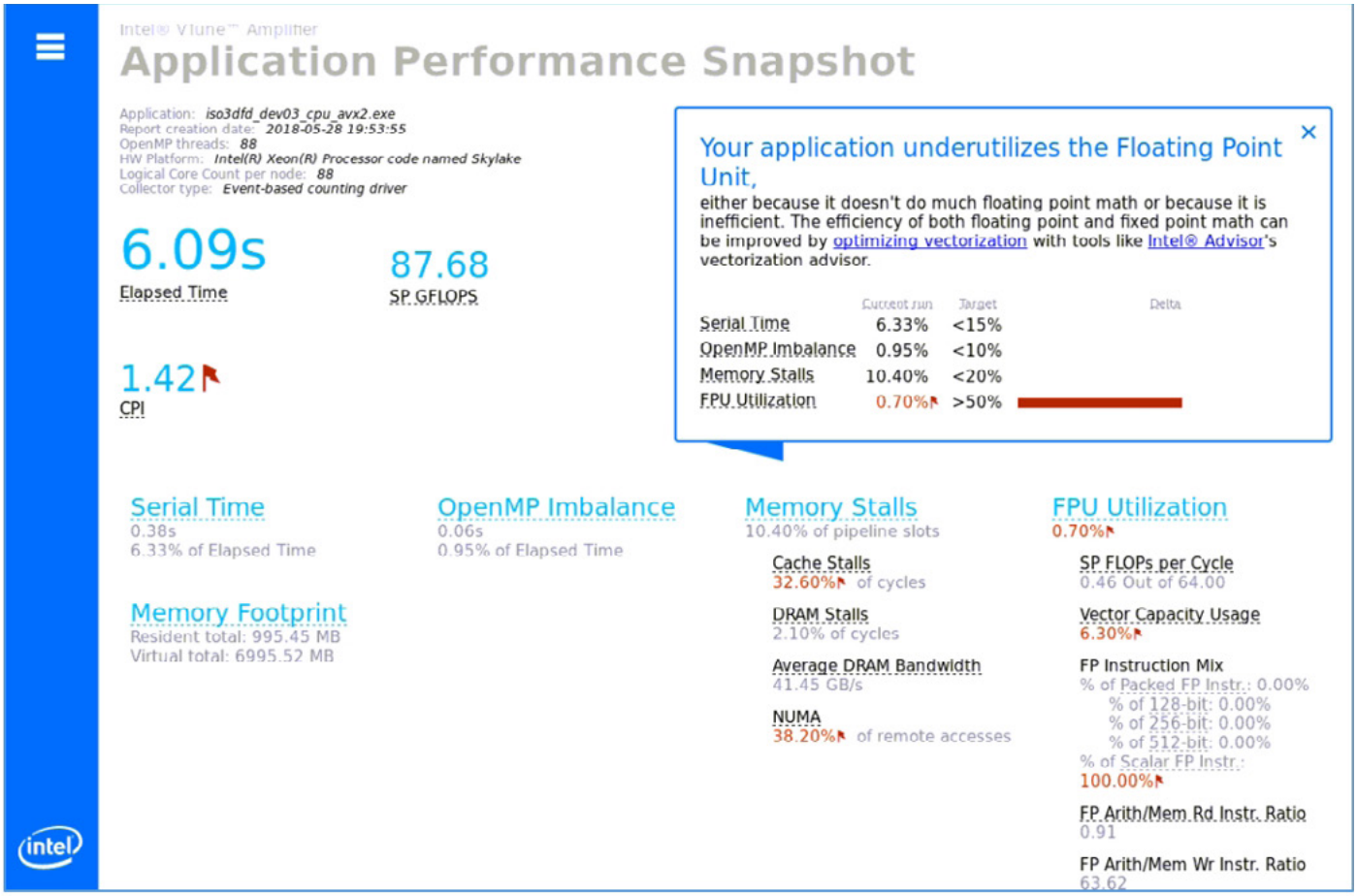
開発者のためにコンピューター・ビジョンとディープラーニングの扉を解放

CHARLOTTE DRYDEN INTEL CORPORATION

Santa Clara で開催された Embedded Vision Summit で、データ視覚化に取り組むイノベーターやリーダーと会えることを楽しみにしていました。

コンピューター・ビジョンとは、人間の目と同じように見えるようにするため機械によって使用される手法とテクノロジーです。これには、デジタル画像をキャプチャー、処理、解析、理解するためのすべてのハードウェアとソフトウェアが含まれます。

この記事の続きはこちら (英語) でご覧になれます。 >



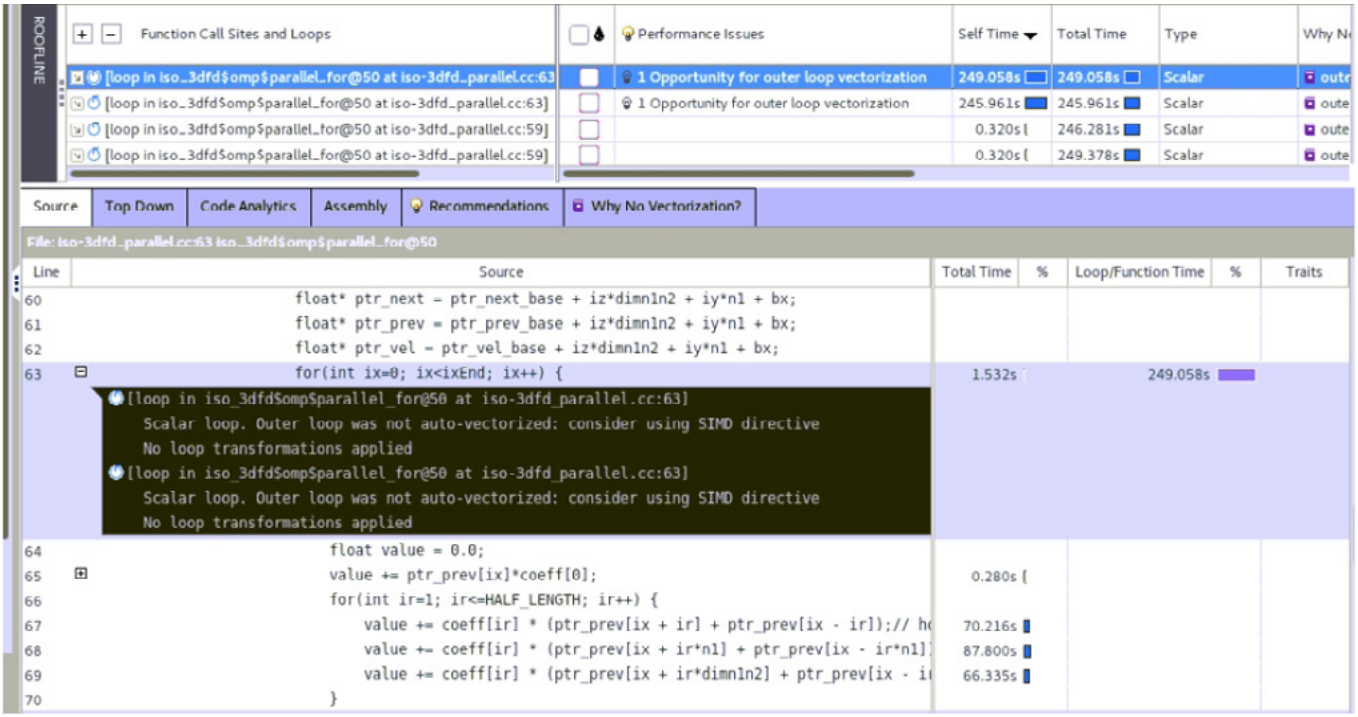
14 **キャッシュ・ブロッキング後の APS レポート**

最適化ステップ 5: ベクトル化を追加する

APS レポートから、カーネルにはスカラー浮動小数点命令のみが含まれていることが分かります ("% of Scalar FP Instr." 値が 100%)。これは、コンパイラーによってコードが自動ベクトル化されていないことを意味します。インテル® Advisor を使用して、コードのベクトル化の可能性を調査します。

```
> advixe-cl -collect survey ./Iso3DFD
```

GUI でレポートを開き、[Survey and Roofline (サーベイ & ルーフライン)] タブを表示します。ハイライト表示したループをコンパイラーがベクトル化できるようにするため、SIMD ディレクティブの使用が推奨されています (**図 15**)。



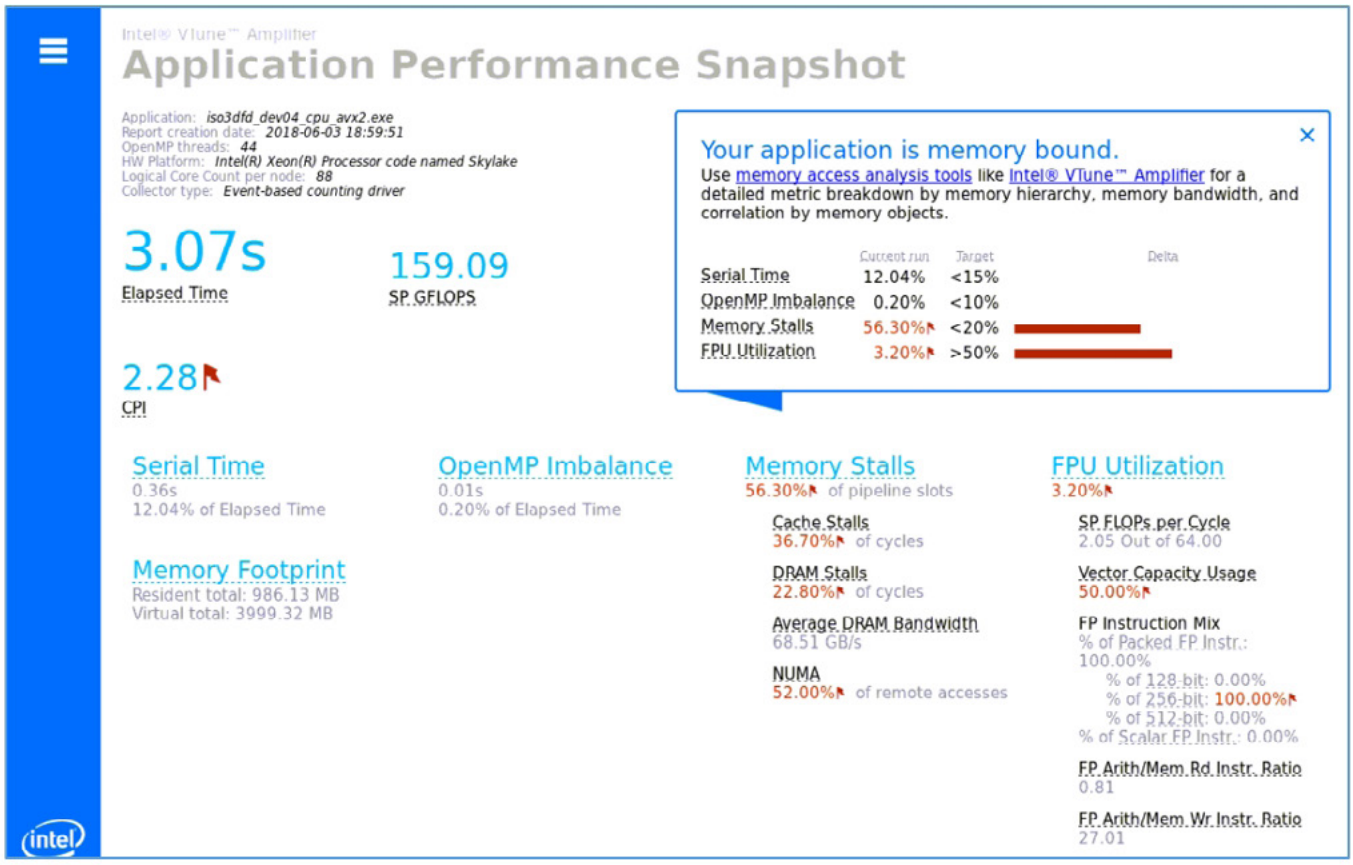
15 **インテル® Advisor はループのベクトル化の可能性を提案**

OpenMP* 標準には最近、ループ反復が独立しているかどうか判断できない場合にコンパイラーによる自動ベクトル化を支援する、**omp simd** プラグマが追加されました。このプラグマを挿入して、自動ベクトル化されるかどうか見てみます (**図 16**)。

```
for(int iz=bz; iz<izEnd; iz++) {
        for(int iy=by; iy<iyEnd; iy++) {
                float* ptr_next = ptr_next_base + iz*dimn1n2 + iy*n1 + bx;
                float* ptr_prev = ptr_prev_base + iz*dimn1n2 + iy*n1 + bx;
                float* ptr_vel = ptr_vel_base + iz*dimn1n2 + iy*n1 + bx;
                #pragma omp simd
                for(int ix=0; ix<ixEnd; ix++) {
                        float value = 0.0;
                        value += ptr_prev[ix]*coeff[0];
                        for(int ir=1; ir<=HALF_LENGTH; ir++) {
                                value += coeff[ir] * (ptr_prev[ix + ir] + ptr_prev[ix - ir]);
                                value += coeff[ir] * (ptr_prev[ix + ir*n1] + ptr_prev[ix - ir*n1]);
                                value += coeff[ir] * (ptr_prev[ix + ir*dimn1n2] + ptr_prev[ix - ir*dimn1n2]);
                        }
                        ptr_next[ix] = 2.0f* ptr_prev[ix] - ptr_next[ix] + value*ptr_vel[ix];
                }
```

16 **omp simd プラグマの挿入**

適切なベクトル命令セットが使用されるように、**-xHost** コンパイラー・オプションを指定してアプリケーションを再コンパイルします。APS レポートは、これによりパフォーマンスが 2 倍向上したことを示しています (**図 17**)。また、ベクトル処理の使用率が 50% になりました。しかし、ベクトル化の効率が向上したことで、メモリーへの負荷が高くなりました。ベクトルユニットへより多くのデータを供給する必要があるため、メモリーストールが再発しています。



17 **ベクトル化後の APS レポート**

インテル® Advisor

現代のハードウェア向けにコードを最適化

詳細

スレッド化、メモリー、ベクトル化の最適化

さらなる最適化 (-qopt-zmm-usage=high コンパイラー・オプションを使用するなど) を試みることができますが、ここではいくつかの基本的なコード変更とインテル® Parallel Studio XE (特に APS) を使用することで、ベースライン・パフォーマンスの 1,767 秒から 3 秒にスピードアップすることができました。APS は、最適化状況を素早くチェックして、チューニングの可能性を特定するのに役立ちました。インテル® VTune™ Amplifier のメモリーアクセス解析やインテル® Advisor のサーベイとメモリー・アクセス・パターンは、特定のパフォーマンス考察の詳細解析に役立ちました。

関連資料

- **インテル® Parallel Studio XE**
- **インテル® Advisor**
- **インテル® VTune™ Amplifier**
- **インテル® VTune™ Amplifier アプリケーション・パフォーマンス・スナップショット (英語)**

BLOG HIGHLIGHTS

ユニバーサル Windows* プラットフォーム・アプリケーションでのインテル® TBB の使用

NIKITA P. INTEL CORPORATION

インテル® スレッディング・ビルディング・ブロック (インテル® TBB) ライブラリーは、C++ アプリケーションを並列化するアルゴリズムのセットを提供します。移植性が高く、インテル® アーキテクチャー・ベースのすべての Windows* デバイスを含むさまざまなプラットフォームをサポートします。

インテル® TBB 2018 では、PC、タブレット、電話、Xbox*、HoloLens*、Surface Hub*、IoT など、あらゆる Windows* デバイスで開発者がアプリケーションを作成して実行できる Windows® 10 エコシステム向けアプリケーション・プラットフォームである、ユニバーサル Windows* プラットフォーム (UWP) がサポートされました。

この記事の続きはこちら (英語) でご覧になれます。>

Java* による不揮発性メモリー・コンピューティング

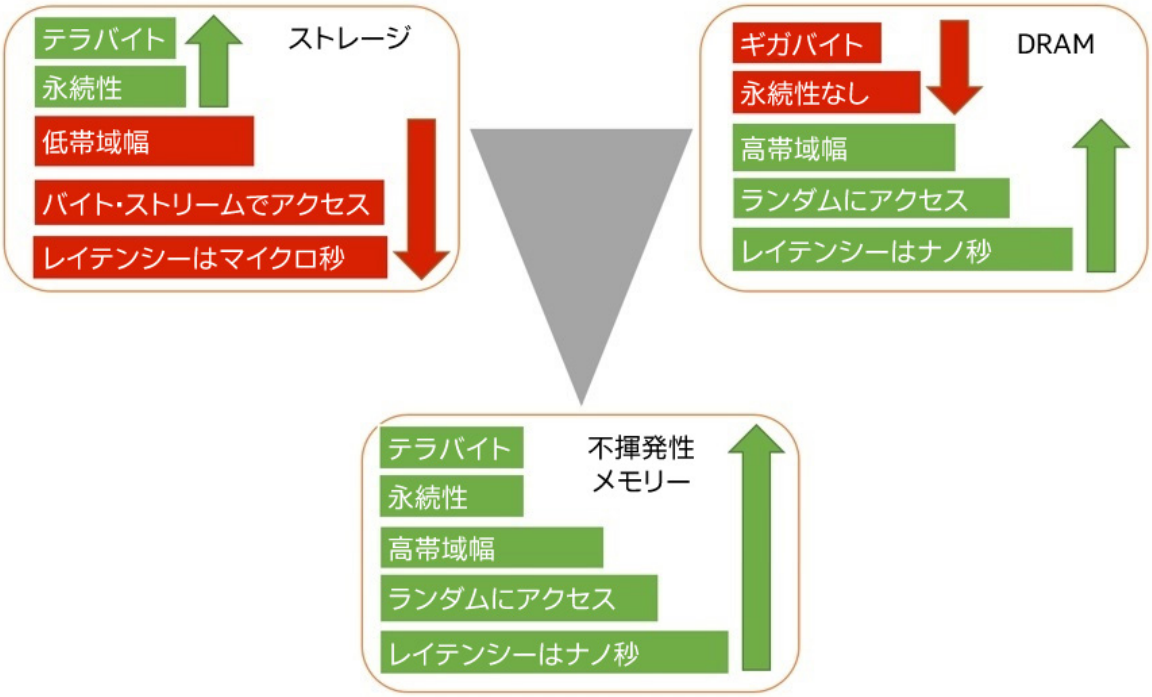
オープンソース・ライブラリーを利用してアプリケーションを不揮発性メモリー対応にする

Eric Kaczmarek インテル コーポレーション ソフトウェア・エンジニアリング・マネージャー
Preetika Tyagi インテル コーポレーション ソフトウェア・エンジニア

コンピューターは大きく発展したにもかかわらず、1960 年代以降、ハードディスクは一貫して、大量のデータを格納する標準的で最適な装置として位置付けられています。ハードディスクは大容量で耐久性に優れていますが、いくつか欠点もあります。帯域幅が低くレイテンシーが大きいため、ディスクから DRAM にデータを転送する際にプロセッサーの待機時間が長くなります。また、バイト・ストリームとしてデータを格納するため、シリアル化と逆シリアル化によるオーバーヘッドが増えます。その結果、データベースのようにデータの永続性が必要な多くのリソース集約型ソフトウェア・アプリケーションにとって、ディスク IO は大きな課題の 1 つとなっています。

コンパイラーの最適化に関する詳細は、**最適化に関する注意事項**を参照してください。

理想的な解決策はデータをメモリーに格納することですが、DRAM は手頃な価格で十分な容量と永続性を提供できません。不揮発性メモリーは、両方の長所を組み合わせて、この課題を解決します (**図 1**)。



1 **ストレージとメモリー**

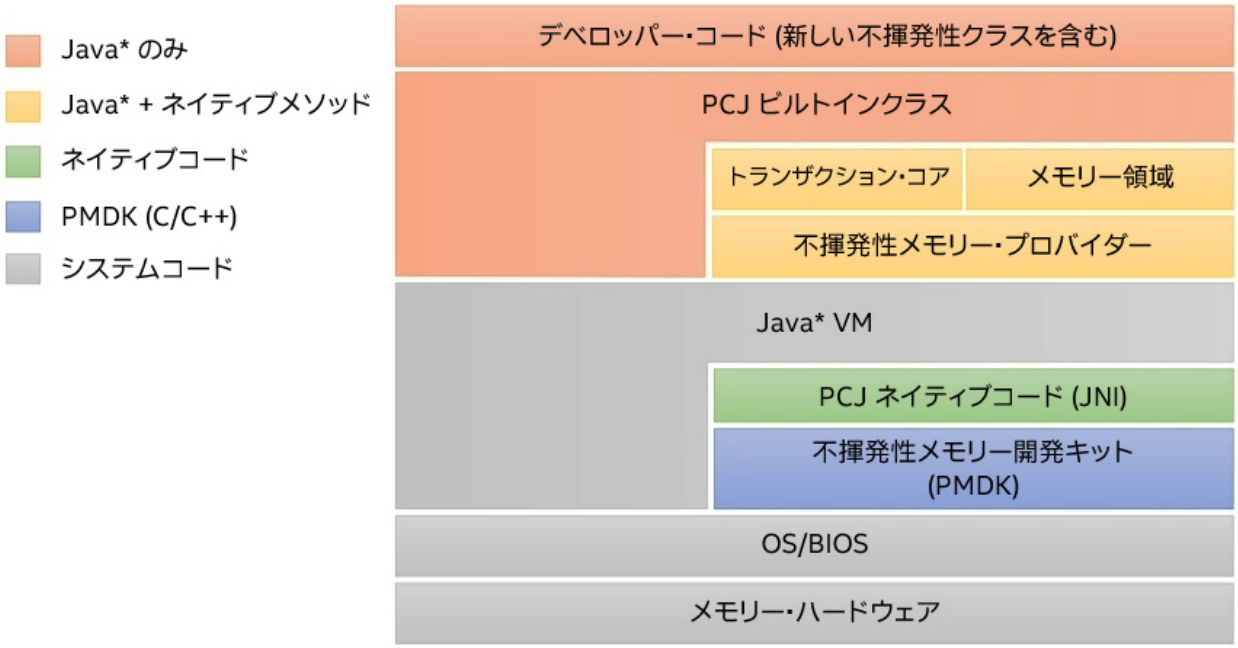
不揮発性メモリーについて

不揮発性メモリーは、永続性と大容量 (テラバイト・レベル) を実現しながら、メモリーと同様のパフォーマンスを提供します。データはマシンを再起動しても保持されます。直接ユーザーモードでアクセスできるため、データパスからカーネル/IO を排除できます。

開発者は、通常のメモリーのように (しかしストレージのように永続的に保持されるものとして)、不揮発性メモリーをプログラムします。**不揮発性メモリー開発キット (PMDK)** (英語) は、不揮発性メモリーを利用するアプリケーションの作成に利用できるオープンソースの C/C++ ライブラリーです。**Java* 向け不揮発性コレクション (PCJ) ライブラリー** (英語) は、Apache Cassandra*、Apache Spark*、Apache Ignite* などの Java* ベースのアプリケーションにも利用範囲を広げます。

Java* 向け不揮発性コレクション (PCJ): 不揮発性メモリー対応 Java* プログラミング・ライブラリー

PCJ ライブラリーは、Java* 開発者がディスク IO の制限を心配することなく不揮発性メモリー対応のアプリケーションを設計または変更できるようにする、オープンソースのパイロット・プロジェクトです。**図 2** に実装スタックを示します。PCJ ライブラリーは、通常の Java* オブジェクトに似た、不揮発性オブジェクトの作成に使用できる不揮発性クラスを提供します。これらのオブジェクトは不揮発性ヒープに格納され、JVM のセッション間やマシンを再起動した場合でも永続的に保持されます。これらの不揮発性オブジェクトの存続期間は到達可能性ベースで、到達不能になるまでガベージ・コレクションされません。データは不揮発性メモリー内にオブジェクト・レイアウト形式で直接格納されるため、シリアル化や逆シリアル化は不要です。PCJ ライブラリーには、通常の Java* クラスに似た、カスタマイズされた不揮発性クラスを定義するための API も用意されています。**図 3** は、PCJ ライブラリーに含まれているビルトイン不揮発性クラスのリストの一部です。



2 **PCJ 実装スタック**

プリミティブ・タイプ （フィールドと配列要素の値、個別のクラスなし）

Boxed プリミティブ （PersistentLong など）

PersistentString

PersistentByteBuffer

PersistentAtomicReference<T extends AnyPersisent>

プリミティブ配列 （PersistentByteArray など、可変および不変）

PersistentArray<E extends AnyPersistent> （可変および不変）

PersistentTuple<T1 extends AnyPersistent, …> （可変および不変）

PersistentArrayList<E extends AnyPersistent>

PersistentHashap<K extends AnyPersistent, V extends AnyPersistent>

PersistentLinkedLIst<E extends AnyPersistent>

PersistentLinkedQueue<E extends AnyPersistent>

PersistentSkipListMap<K extends AnyPersistent, V extends AnyPersistent>

PersistentSkipTreeMap<K extends AnyPersistent, V extends AnyPersistent>

PersistentSkipHashMap<K extends AnyPersistent, V extends AnyPersistent>

ObjectDirectory は <String, T extends AnyPersistent> の無制限に到達可能なルートマップ

3 不揮発性クラス

PCJ によるプログラミング

図 4 は、PCJ ライブラリーを使用して不揮発性メモリーにデータを割り当てて格納するコードの一部です。

```
PersistentIntArray a = new PersistentIntArray(1024); // Ints are allocated on persistent heap
a.set(0, 123);                                       // 4-byte int value written to persistent heap
a = null;                                            // Array is unreachable. Object will be collected
PersistentIntArray data = new PersistentIntArray(1024);
ObjectDirectory.put("Application_data", data);       // no serialization, reference to array is written
data.set(0, 123);

// reboot
PersistentIntArray data1 = ObjectDirectory.get("Application_data", PersistentIntArray.class);
assert (data1.get(0) == 123);                        // data1.get(0) still has '123' value after reboot
```

4 **再起動後のデータの永続性**

図 5 は通常の Java* クラス、**図 6** はその不揮発性バージョンです。

```
public final class Employee {
    private final long id;
    private String name;

    public Employee(long id, String name) {
        this.id = id;
        setName(name);
    }

    public long getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int hashCode() {
        return Long.hashCode(getId());
    }

    public boolean equals(Object obj) {
        if (!(obj instanceof Employee)) return false;
        Employee emp = (Employee) obj;
        return emp.getId() == getId() && emp.getName().equals(getName());
    }

    public String toString() {
        return String.format("Employee (%d,%s))", getId(), getName());
    }
}
```

5 **通常の Java* クラス**

```java
public class Employee extends PersistentObject {
    private static final LongField ID = new LongField();
    private static final StringField NAME = new StringField();
    private static final ObjectType<Employee> TYPE = ObjectType.withFields(Employee.class, ID, NAME);

    public Employee(long id, PersistentString name) {
        super(TYPE);
        setLongField(ID, id);
        setName(name);
    }

    private Employee(ObjectPointer<Employee> p) {
        super(p);
    }

    public long getId() {
        return getLongField(ID);
    }

    public PersistentString getName() {
        return getObjectField(NAME);
    }

    public void setName(PersistentString name) {
        setObjectField(NAME, name);
    }

    public int hashCode() {
        return Long.hashCode(getId());
    }

    public boolean equals(Object obj) {
        if (!(obj instanceof Employee)) return false;
        Employee emp = (Employee) obj;
        return emp.getId() == getId() && emp.getName().equals(getName());
    }

    public String toString() {
        return String.format("Employee(%d, %s)", getId(), getName());
    }
}
```

6 不揮発性クラス

PCJ ライブラリーはトランザクションのサポートも提供します。PCJ ライブラリーのメソッドはすべてトランザクショナルです。不揮発性メモリーに対する変更はすべて行われるか、全く行われません。トランザクショナルな性質を複数のメソッド呼び出しに拡張する場合は、トランザクションに記述します (**図 7**)。

Java* 開発者は、PCJ ライブラリーを利用して、自然な方法で大量の不揮発性データを操作するハイパフォーマンス・アプリケーションを記述できます。新しいアプリケーションは設計段階で PCJ の機能を追加できますが、既存のアプリケーションは不揮発性メモリーを使用するように変更する必要があります。

```
PersistentArrayList<PersistentString> movies = new PersistentArrayList<>();
PersistentArrayList<PersistentString> movieIndex = new PersistentArrayList<>();

public void addMovie(PersistentString movie) {
    Transaction.run(() -> {
        movies.add(movie);
        movieIndex.add(movie);
    });
}
```

7 **PCJ によるトランザクション**

低水準不揮発性ライブラリー (LLPL)

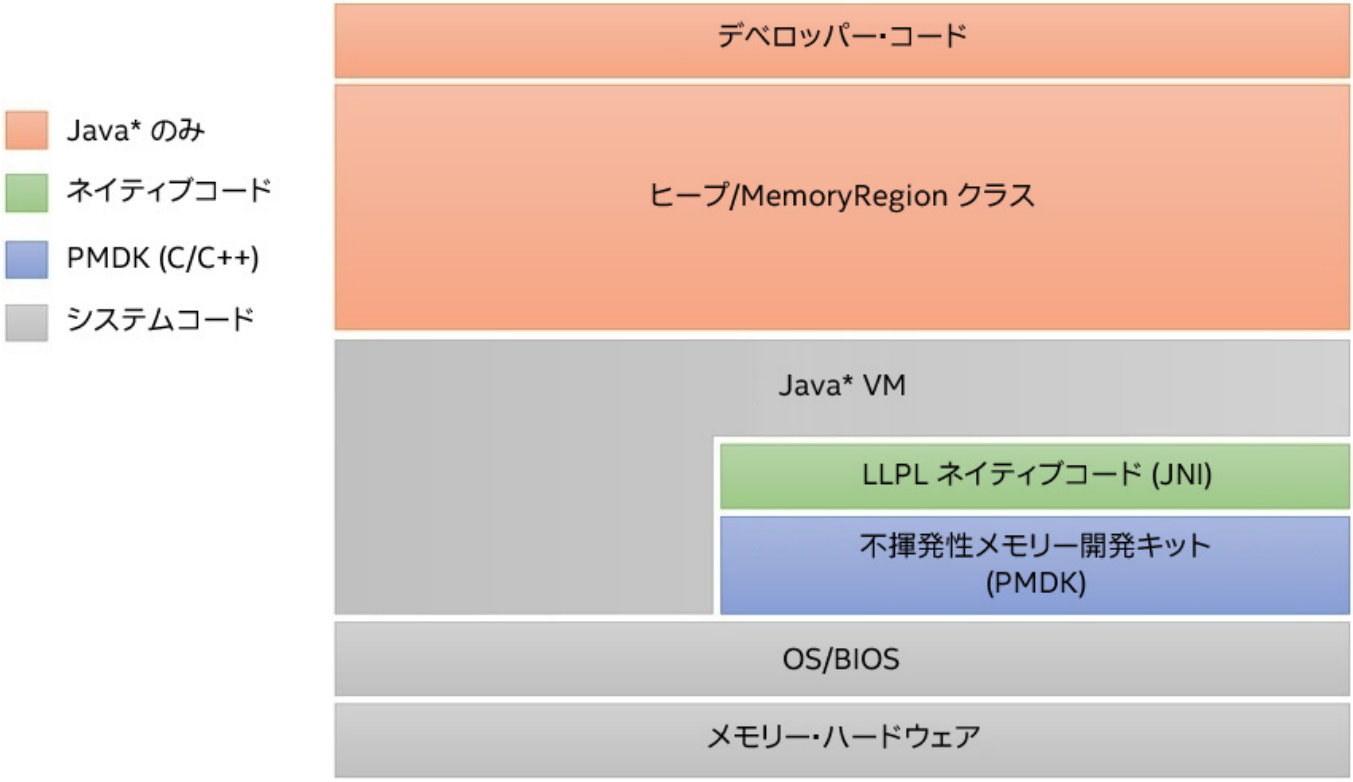
PCJ ライブラリーは、不揮発性メモリーの任意の領域に対する個別の低水準アクセスも提供します。この優れた柔軟性により、Java* 開発者は、独自のデータ抽象化を作成できます。**図 8** に LLPL の実装スタックを示します。ヒープ API は MemoryRegion の割り当てと解放を行います。MemoryRegion インターフェイスは、不揮発性メモリーにアクセスする setter と getter を提供します。3 種類の MemoryRegions が提供されます。

1. **MemoryRegion<Raw>:** 揮発性の用途または呼び出し元が提供するデータの一貫性に適しています。
2. **MemoryRegion<Flushable>:** フェイルセーフな `flush()` メソッドと `isFlushed()` メソッドを含みます。
3. **MemoryRegion<Transactional>:** 書き込みはトランザクショナルです。

LLPL を使用すると、開発者は要件に最も適した実装を選択し、低水準で既存のアプリケーションを変更できます。

不揮発性メモリーに対応した Apache Cassandra*

Wikipedia* によれば、「**Apache Cassandra*** は、多くのコモディティー・サーバーで大量のデータを制御するように設計された、**単一障害点**のない高可用性を提供する、**無料のオープンソースの分散型 NoSQL データベース**管理システムです。」Apache Cassandra* は、多くの Web サービスで利用されています (**図 9**)。



8 **LLPL 実装スタック**

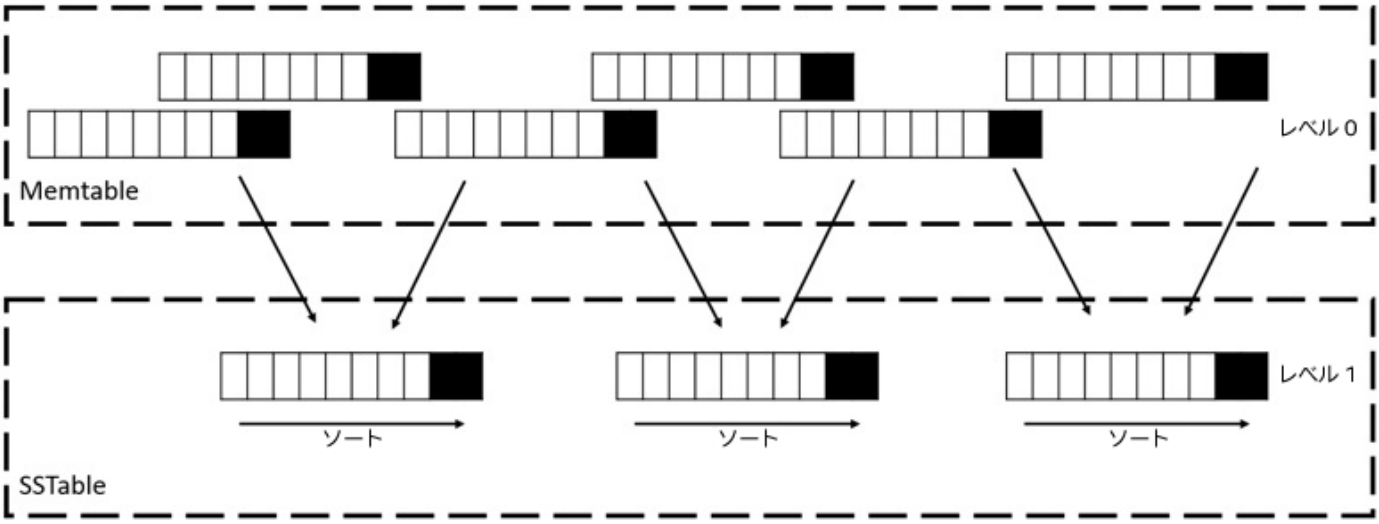
9 **Apache Cassandra* のユーザー**

現在、Cassandra* はデータストレージとして HDD/SSD を使用しています。パフォーマンスを向上するため、さまざまな最適化手法を利用しています。

- キャッシュ
- インデックス / オフセット
- データサマリー
- ブルームフィルター

大量のディスク IO はスループットを大幅に低下させ、高負荷状態のレイテンシーが増加するため、これらの最適化の主な目的は、ディスク IO の回数を減らすことです。不揮発性メモリーは、ディスクストレージ要件を排除します。インテルは、PCJ ライブラリーによりソフトウェア・スタック全体をどのように単純化できるか実証するため、Cassandra* の概念実証バージョンを開発しました。このライブラリーを使用すると、不揮発性メモリーに直接オブジェクト・レイアウト形式でデータを格納できます。

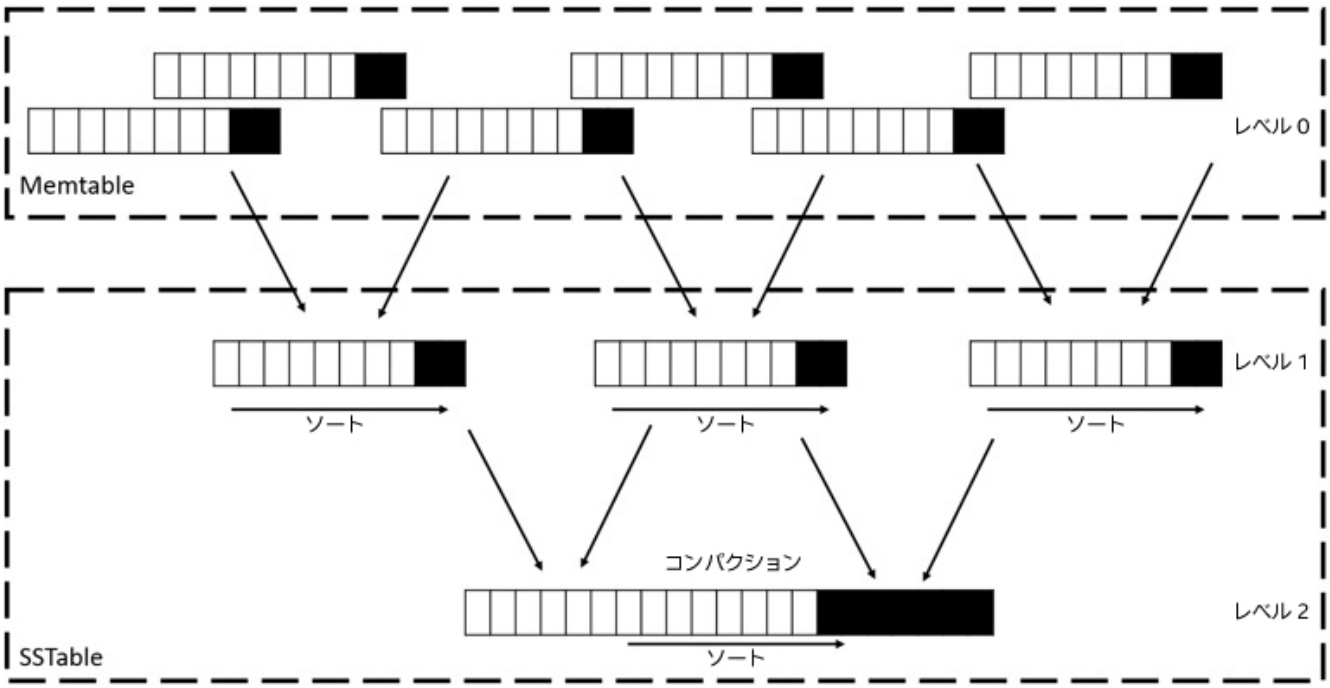
Cassandra* の既存のストレージメカニズムは、キー / バリュー型でデータを構成するため、複数のレベルを使用するログ構造化マージツリー (LSM ツリー) データ構造に基づいています。各レベルのデータはキーでソートされ、マージソートによりレベル間を移動します。各キーは列と関連付けられます。**図 10** は、Cassandra* の 2 つのレイヤーを含む LSM ツリーの単純なバージョンです。レベル 0 の Memtable (フルになりディスクにフラッシュされるまでデータを保持するインメモリー・バッファー) は DRAM に格納されます (データはキーでソートされます)。Memtable がしきい値に達すると、データはレベル 1 のソートされた文字列テーブル形式 (SSTable、Memtable フラッシュから生成されるオンディスクの不変のファイル) でディスクにフラッシュされます。



10 **既存の Cassandra* ストレージメカニズム**

SSTable は不変であるため、同じキーの複数の書き込みの結果として、複数の SSTable に列の異なるバージョンを保持できます。このスキームでは、データが Memtable に書き込まれると、オンディスクのレベル 1 ストレージにフラッシュされるのを待つことなく、クライアントに直ちにレスポンスを返すことができるため、書き込み操作は効率的です。しかし、クライアントに最新の正確なレスポンスを返すには複数のオンディスクの SSTable を読み取る必要があるため、同じ列の複数のバージョンを保持する SSTable の数が増加すると、読み取りのレイテンシーは大きくなります。また、古くなったデータがディスクに存在し続けるため、ディスクの利用効率は低下します。

読み取りパフォーマンスの低下を軽減し、ディスクを効率的に再利用するため、Cassandra* はコンパクションと呼ばれるバックグラウンド・プロセスを使用しています。コンパクションの主な目的は、複数の SSTable を 1 つの新しい SSTable にマージすることです (**図 11**)。カラムを組み合わせてデータの最新のコピーを保持することにより、キーをマージします。また、トゥームストーン (削除されたことを示すキーに書き込まれる特別な値) を削除して、新しい SSTable 全体のデータのサイズを減らします。Cassandra* では、反復的にコンパクションを実行することで、読み取りレイテンシーとディスク容量の両方を調節します。しかし、コンパクションは CPU 負荷の高いタスクであるため、予想外にレイテンシーが急増することがあります。



11 **コンパクション・プロセス**

PCJ ライブラリーを使用して開発された、不揮発性メモリーに対応した Cassandra* プロトタイプは、可変データ構造 (PMTable) を不揮発性メモリーに格納できます。シリアル化されたオンディスクの SSTable とは対照的に、可変 PMTable はオブジェクトとフィールド形式でデータを直接格納できます (**図 12**)。次のようないくつかの利点があります。PMTable は可変であるため、データはインプレースで書き込まれ / 更新され、最新バージョンの列のみ保持されます。そのため、データ読み取りのレイテンシーは一貫しています。データはオブジェクト・レイアウト形式で格納されるため、シリアル化と逆シリアル化のオーバーヘッドはありません。キーは、ディスク IO オーバーヘッドのない不揮発性メモリーに存在する PMTable のデータを調べるために使用できます。このスキームでは、インプレース更新によりデータのマージが必要ないため、コンパクションのオーバーヘッドも排除されます。ディスクベースの最適化機能 (ブルームフィルター、インデックス、キャッシュなど) は不要になるため、読み取りパスが単純化され、異常なレイテンシーを予測しやすくなります。現在のプロトタイプは、機能が制限されています。インテルは、このプロトタイプをオープンソースにして完全に機能するように、Apache* コミュニティーと協力して作業を行っています。オープンソースの PCJ ライブラリーは、Java* 開発者がアプリケーションを不揮発性メモリーに対応できる、使いやすい不揮発性クラスを提供します。

12 **PMTable (不揮発性メモリーテーブル)**

参考資料およびオープンソース・プロジェクト (英語)

- **Java* 向け不揮発性コレクション (PCJ)**
- **不揮発性メモリー開発キット (PMDK)**
- **Apache Cassandra***
- **不揮発性メモリー・プログラミング入門**

ビジョンを
コード化しよう

intel Software

コンピューター・ビジョンと
ディープラーニング推論を、
スマートカメラや
ビデオ監視から
ロボット工学や
交通まで、
さまざまな
アプリケーションに
迅速かつ容易に統合。
新しい OpenVINO™
ツールキットでエッジから
クラウドまでワークロードを加速。

無料のダウンロード (英語) >

コンパイラーの最適化に関する詳細は、最適化に関する注意事項 (software.intel.com/en-us/articles/optimization-notice#opt-jp) を参照してください。
Intel、インテル、Intel ロゴ、OpenVINO は、アメリカ合衆国および / またはその他の国における Intel Corporation の商標です。
© Intel Corporation.

より高速な勾配ブースティング決定木

インテル® DAAL によりマシンラーニングのパフォーマンスを向上する方法

Ying Hu インテル コーポレーション テクニカル・コンサルティング・エンジニア
Oleg Kremnyov 同ソフトウェア・デベロッパー
Ivan Kuzmin 同ソフトウェア開発マネージャー

マシンラーニング (ML) は、この 10 年の間に爆発的に成長しました。画像や音声認識のような特定の分野とともに、ディープラーニングが成功を収めた分野で、広範な実際の ML 問題 (回帰や分類) を解くための最も人気のある手法の 1 つは勾配ブースティングです。最近では、ブースティングは商用 Web 検索エンジンのページランキングや ML チャレンジのソリューションにも使用されています。

インテル® データ・アナリティクス・アクセラレーション・ライブラリー (インテル® DAAL) は、インテル® プロセッサー向けに最適化された実用的な ML アルゴリズムのコレクションを提供します。最新バージョンには、確率的勾配ブースティング手法を使用した勾配ブースティング木分類アルゴリズムと回帰アルゴリズムが含まれています。この実装は、マルチスレッド化とベクトル化により、パフォーマンスを向上し機能を強化します。

コンパイラーの最適化に関する詳細は、**最適化に関する注意事項**を参照してください。

この記事では、勾配ブースティング木アルゴリズムの概要と ML の世界で広く利用されている理由を説明します。インテル® DAAL の勾配ブースティングは、人気のある **XGBoost** (英語) ライブラリーと比較して、同じデータセットの訓練パフォーマンスで最大 6.5 倍高速です。

勾配ブースティングと決定木

勾配ブースティングは、回帰、分類、ランキングのようなさまざまなタスクで最高水準の精度を達成するために使用される強力な ML 教師ありアルゴリズムです。Jerome Friedman 氏により提唱された一般化勾配ブースティング・アルゴリズム[1] は、評価された予測関数の分散を縮小するためにアンサンブルまたはコミッティー手法を使用します。勾配ブースティングは、一般にベース学習器として決定木を使用します。

決定木は、セグメンテーション、回帰、分類などのさまざまな分析ソリューションで使用される、単純で強力な分析モデルです。[**編集者注 :** 決定木モデルの長所、構文、スコアリング、制限と解決方法は、**この記事** (英語) を参照してください。] アンサンブル・ラーニングは、決定木のサンプリングや過剰適合の問題を管理する高度な手法の 1 つです。ランダムフォレストのように、もう 1 つの人気のある木アンサンブル・モデルは勾配ブースティング木で、scikit-learn、XGBoost、LightGBM、GBM、H20、Spark* MLlib、OpenCV* を含む、多くの ML ソフトウェア・パッケージに実装されています。

勾配ブースティング決定木アルゴリズムの詳細

前述のとおり、勾配ブースティング決定木 (GBDT) 分類アルゴリズムと回帰アルゴリズムは、確率的勾配ブースティング手法を使用して構築された回帰 (決定) 木のアンサンブル処理です。[3]

n p 次元の特徴ベクトルの n 特徴ベクトル $X = \{x_1 = (x_{11},\cdots,x_{1p}), \ldots, x_n = (x_{n1},\cdots,x_{np})\}$ および n 応答 $Y = \{y_1,\cdots,y_n\}$ について、アルゴリズムの学習プロセスは、特徴と応答データセットに基づいた勾配ブースティング木分類または回帰モデルを作成し、分類と回帰モデルを使用して新しいサンプルを分類 / 予測します。

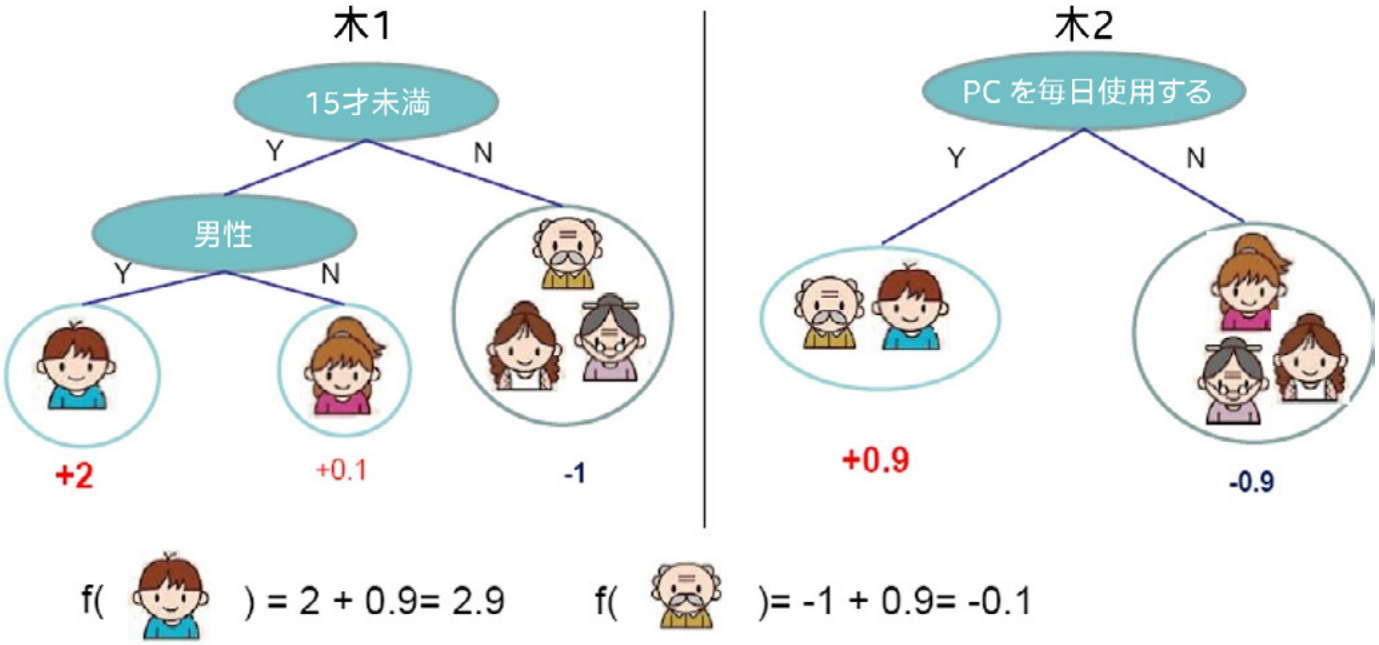
木アンサンブル・モデルは、M 加法関数を使用して出力 $(\hat{y}_i) = f(x) = \sum m/_{k=1} f_k(x_i), f_k \in F$ を予測します。ここで、$F = \{f(x)=w_{q(x)}, q: R^p \rightarrow T, w \in R^T\}$ は回帰木の空間、T は木の葉の数、w は葉の重みベクトル、w_i は i 番目の葉のスコアです。$q(x)$ は、観測を対応する葉インデックスにマップする各木の構造を表します。

訓練プロシージャーは、負の勾配方向をポイントする関数 (回帰木) を反復して選択することにより目的関数を最小化する、反復関数勾配降下法アルゴリズムです。目的関数は次のようになります。

$$L(f) = \sum_{i=1}^{n} l(y_i, f(x_i)) + \sum_{k=0}^{M} \Omega(f_k)$$

ここで、**l(f)** は 2 回微分可能な凸損失関数、**Ω(f) = γT+ 1/2 λ||w||** は葉の数 **T** および各木の重みの L2 ノルム **||w||** により定義されたモデルの複雑さにペナルティーを課す正則化項、**γ** および **λ** は正則化パラメーターです。

GBDT アルゴリズムの原理は詳細に説明されています。[7] 労力を要する処理は木の構築です。**図 1** は、その人がコンピューター・ゲームを好きかどうか判断する付加的な木構造です。木の葉 (ターミナル) ノードは、対応する応答値 (木からの予測結果) を表します。(詳細は、**こちら** (英語) を参照してください。)



1 **木アンサンブル**[6]

訓練フロー

S = (X, Y) を観察点のセットとします。反復数 **M**、損失関数 **l(f)**、回帰木訓練パラメーター、正則化パラメーター **γ** および **λ**、shrinkage (学習率) パラメーター **θ** などの訓練パラメーターで、アルゴリズムは次の操作を行います。

- 最初の推測 $(\hat{y}_i)^{(0)}$, **i=1,..n** を見つけます。
- **k = 1,..., M** で繰り返します。
  - — g_i および h_i, **i =1,...n** を更新します。
  - — $-\frac{1}{2}\sum_{j=1}^{T}\frac{G_j^2}{H_j+\lambda}+\gamma T$ ここで、$G_j=\sum_{i\in I_j} g_j, H_j=\sum_{i\in I_j} h_j, I_j=\{i|\, q(x_i)=j\}, j=1,..T$
  - — 最適な重み $w_j^* = \frac{G_j}{H_j+\lambda}$ を葉 **j,j = 1,...T** に割り当てます。
  - — shrinkage パラメーター **θ** を木の葉に適用して木をモデルに追加します。
  - — $\hat{y}_1^{(k)}$ を更新します。

木を成長させるアルゴリズムは次のとおりです。

- 必要に応じて (確率的勾配ブースティング)、ブートストラップ訓練セットを生成します。
- 深さ 0 の木から開始します。
- 木の葉ノードについてそれぞれ、次の操作を行います。

- 必要に応じて、分割を見つけるための特徴のサブセットを選択します (確率的勾配ブースティング)。
- 利得が最大になる分割を見つけます。

$$\frac{G_L^2}{H_L+\lambda}+\frac{G_R^2}{H_R+\lambda}-\frac{(G_L+G_R)^2}{H_L+H_R+\lambda}-\gamma$$

- 終了基準が満たされたら停止します。[6]

予測フロー

木が訓練段階で構築されたら、構築された木を使用して指定されたクエリーのセットについて予測を行います。分類アルゴリズムは、各クラスについてすべての木の応答の和を計算し、応答値が最大のクラス (最高のクラス確率) で予測を判断します。回帰アルゴリズムは、集計結果を指定されたサンプルの最終結果として返します。

インテル® DAAL の GBDT 実装

GBDT アルゴリズムは (特に連続特徴と大規模なデータセットで) 計算負荷が高くなりますが、インテル® DAAL は分類と回帰向けに高度にチューニングされた実装を提供しています。最大限のパフォーマンスを引き出せるように、この実装では、木の構築と予測でベクトル化および複数レベルの並列化を使用しています。

インテル® DAAL で実装されている訓練アルゴリズムは、木を構築する 2 種類の分割計算モードをサポートします。

1. **exact** - 特徴の最良の分割を検索するときにすべての可能な分割値が調査されます。
2. **inexact** - 連続特徴は離散ビンにバケットされ、バケットの境界のみ調査されます。バケットにより、各ステップで計算する分割数が減るため、計算は速くなります。

図 2 は、インテル® DAAL を使用して GBDT モデルを訓練するサンプルコードです。

```
trainDatasetFileName = 'df_classification_train.csv'
nFeatures = 30
nClasses = 6

# 勾配ブースティング木のパラメーター
maxIterations = 50
minObservationsInLeafNode = 1

# 勾配ブースティング木分類アルゴリズムのモデル･オブジェクト
model = None
predictionResult = None
testGroundTruth = None

def trainModel():
    # FileDataSource<CSVFeatureManager> を初期化して .csv ファイルから入力データを取得
    trainDataSource = FileDataSource(
                                    trainDatasetFileName,
                                    DataSourceIface.notAllocateNumericTable,
                                    DataSourceIface.doDictionaryFromContext)
    global model

    # 訓練データとラベルの数値テーブルを作成
    trainData = HomogenNumericTable(nFeatures, 0, NumericTableIface.notAllocate)
    trainGroundTruth = HomogenNumericTable(1, 0, NumericTableIface.notAllocate)
    mergedData = MergedNumericTable(trainData, trainGroundTruth)

    # 入力ファイルからデータを取得
    trainDataSource.loadDataBlock(mergedData)

    # 勾配ブースティング木分類モデルを訓練するアルゴリズム･オブジェクトを作成
    algorithm = training.Batch(nClasses)
    algorithm.parameter().maxIterations = maxIterations
    algorithm.parameter().minObservationsInLeafNode = minObservationsInLeafNode
    algorithm.parameter().featuresPerNode = nFeatures

    # 訓練データセットと関連する値をアルゴリズムに渡す
    algorithm.input.set(classifier.training.data, trainData)
    algorithm.input.set(classifier.training.labels, trainGroundTruth)

    # モデルを訓練して訓練アルゴリズムの結果を取得
    trainingResult = algorithm.compute()

    model = trainingResult.get(classifier.training.model)
```

2 **GBDT モデルの訓練**

splitMethod、maxIterations、maxTreeDepth、shrinkage などの訓練パラメーターおよびデフォルト値は、「**使用モデル：訓練と予測**」(英語) を参照してください。

図 3 は、訓練済み GBDT モデルを使用して予測を行うサンプルコードです。

```
testDatasetFileName = 'df_classification_test.csv'

def testModel():
   global testGroundTruth, predictionResult
   # FileDataSource<CSVFeatureManager> を初期化して .csv ファイルからテストデータを取得
   testDataSource = FileDataSource(
                                   testDatasetFileName,
                                   DataSourceIface.notAllocateNumericTable,
                                   DataSourceIface.doDictionaryFromContext)

   # テストデータとラベルの数値テーブルを作成
   testData = HomogenNumericTable(nFeatures, 0, NumericTableIface.notAllocate)
   testGroundTruth = HomogenNumericTable(1, 0, NumericTableIface.notAllocate)
   mergedData = MergedNumericTable(testData, testGroundTruth)

   # 入力ファイルからデータを取得
   testDataSource.loadDataBlock(mergedData)

   # 勾配ブースティング木分類予測のアルゴリズム･オブジェクトを作成
   algorithm = prediction.Batch(nClasses)

   # テスト･データセットと訓練済みモデルをアルゴリズムに渡す
   algorithm.input.setTable(classifier.prediction.data, testData)
   algorithm.input.setModel(classifier.prediction.model, model)

   # 予測の結果を計算してアルゴリズムの結果を取得
   # (classifier.prediction の結果クラス)
   predictionResult = algorithm.compute()
```

3 **訓練済み GBDT モデルを使用して予測を行う**

パフォーマンスに関する考察と結果

XGBoost (英語) は、木ブースティング向けに設計され最適化されたライブラリーです。[8] さまざまな ML ソフトウェア・パッケージに実装されており、C++、R、Python*、Julia*、Java* ユーザー向けのネイティブ・インターフェイスを提供します。ここでは、インテル® DAAL と XGBoost を比較しました。このパフォーマンスの比較はテストの一部を示していることに注意してください。どちらのパッケージもテスト時点で最新のバージョンを使用しました。**表 1** は、インテル® DAAL 2019 Beta と XGBoost 1.6 のパフォーマンスの比較を示しています。結果は、4 つの異なるデータセット (ML リポジトリーの HIGGS、Letter、ISOLET、MNIST) の分類を測定したものです。

- **HIGGS** (英語): ヒッグス粒子を生成するシグナルプロセスと生成しないバックグラウンド・プロセスを区別する 2 つの分類問題。サンプル数は 100 万で、それぞれ 28 の特徴を含みます。
- **Letter** (英語): 文字画像の特徴のデータベース。目標は、26 の英語の大文字を識別することです。サンプル数は 16,000 で、それぞれ 16 の特徴を含みます。
- **ISOLET** (英語): どの文字名が話されたか予測する単純な分類タスク。サンプル数は 7,797 で、それぞれ 617 の特徴を含みます。
- **MNIST** (英語): 手書きの数字を識別する標準の分類タスク。訓練セットは手書きの数字の 60,000 枚の画像、テストセットは 10,000 枚の画像でそれぞれ構成されています。NIST から入手できる大規模なセットのサブセットです。数字はサイズで正規化され、固定サイズの画像にされます。

テストは、50 の木を使用して、デフォルトのインテル® DAAL のパラメーター (inexact モデル) と XGBoost (近似モデル) で行われました。結果は、平均で、インテル® DAAL が訓練で 2.85 倍 (最大 6.5 倍)、推論で約 1.2 倍高速になりました。

表 1. インテル® DAAL と XGBoost の比較

	HIGGS	MNIST	Letter	ISOLET
訓練 (秒)				
XGBoost	22.6	119.5	3.9	15
インテル® DAAL	10.5	95.7	0.6	9.98
スピードアップ	2.15	1.25	6.5	1.5
推論 (ミリ秒)				
XGBoost	161	29.7	26.7	24
インテル® DAAL	150	25.7	31	15
スピードアップ	1.07	1.16	0.86	1.6

システム構成 : ハードウェア : インテル® Xeon® Silver 4110 プロセッサー、2x8 コア、2.10GHz、32GB RAM。オペレーティング・システム : Ubuntu* 16.04 LTS。ソフトウェア : インテル® DAAL 2019 Beta。

性能に関するテストに使用されるソフトウェアとワークロードは、性能がインテル® マイクロプロセッサー用に最適化されていることがあります。SYSmark* や MobileMark* などの性能テストは、特定のコンピューター・システム、コンポーネント、ソフトウェア、操作、機能に基づいて行ったものです。結果はこれらの要因によって異なります。製品の購入を検討される場合は、他の製品と組み合わせた場合の本製品の性能など、ほかの情報や性能テストも参考にして、パフォーマンスを総合的に評価することをお勧めします。詳細については、www.intel.com/benchmarks (英語) を参照してください。ベンチマークの出典 : インテル コーポレーション

最適化に関する注意事項 : インテル® コンパイラーでは、インテル® マイクロプロセッサーに限定されない最適化に関して、他社製マイクロプロセッサー用に同等の最適化を行えないことがあります。これには、インテル® ストリーミング SIMD 拡張命令 2、インテル® ストリーミング SIMD 拡張命令 3、インテル® ストリーミング SIMD 拡張命令 3 補足命令などの最適化が該当します。インテルは、他社製マイクロプロセッサーに関して、いかなる最適化の利用、機能、または効果も保証いたしません。本製品のマイクロプロセッサー依存の最適化は、インテル® マイクロプロセッサーでの使用を前提としています。インテル® マイクロアーキテクチャーに限定されない最適化のなかにも、インテル® マイクロプロセッサー用のものがあります。この注意事項で言及した命令セットの詳細については、該当する製品のユーザー・リファレンス・ガイドを参照してください。
注意事項の改訂 #20110804

パフォーマンスの向上

インテル® DAAL には、インテル® プロセッサー向けに最適化された ML アルゴリズムのコレクションが含まれています。最近の更新で、勾配ブースティング木分類アルゴリズムと回帰アルゴリズムが追加されました。インテル® DAAL 勾配ブースティングは、訓練と推論の両方のモデルで、人気のある XGBoost ライブラリーを上回るパフォーマンスを達成しました。

参考資料 (英語)

1. Jerome H. Friedman. "**Greedy Function Approximation: A Gradient Boosting Machine**," Annals of Statistics, 1999.
2. **Kaggle: The Home of Data Science and Machine Learning**
3. **インテル® DAAL 2018 Update 3 デベロッパー・ガイド**
4. **XGBoost on GitHub**
5. **インテル® DAAL ランダムフォレスト**
6. Tianqi Chen. "**Introduction to Boosted Trees**," University of Washington, 2014.
7. Tianqi Chen and Carlos Guestrin. "**XGBoost: A Scalable Tree Boosting System**," 22nd SIGKDD Conference on Knowledge Discovery and Data Mining, 2016.

BLOG HIGHLIGHTS

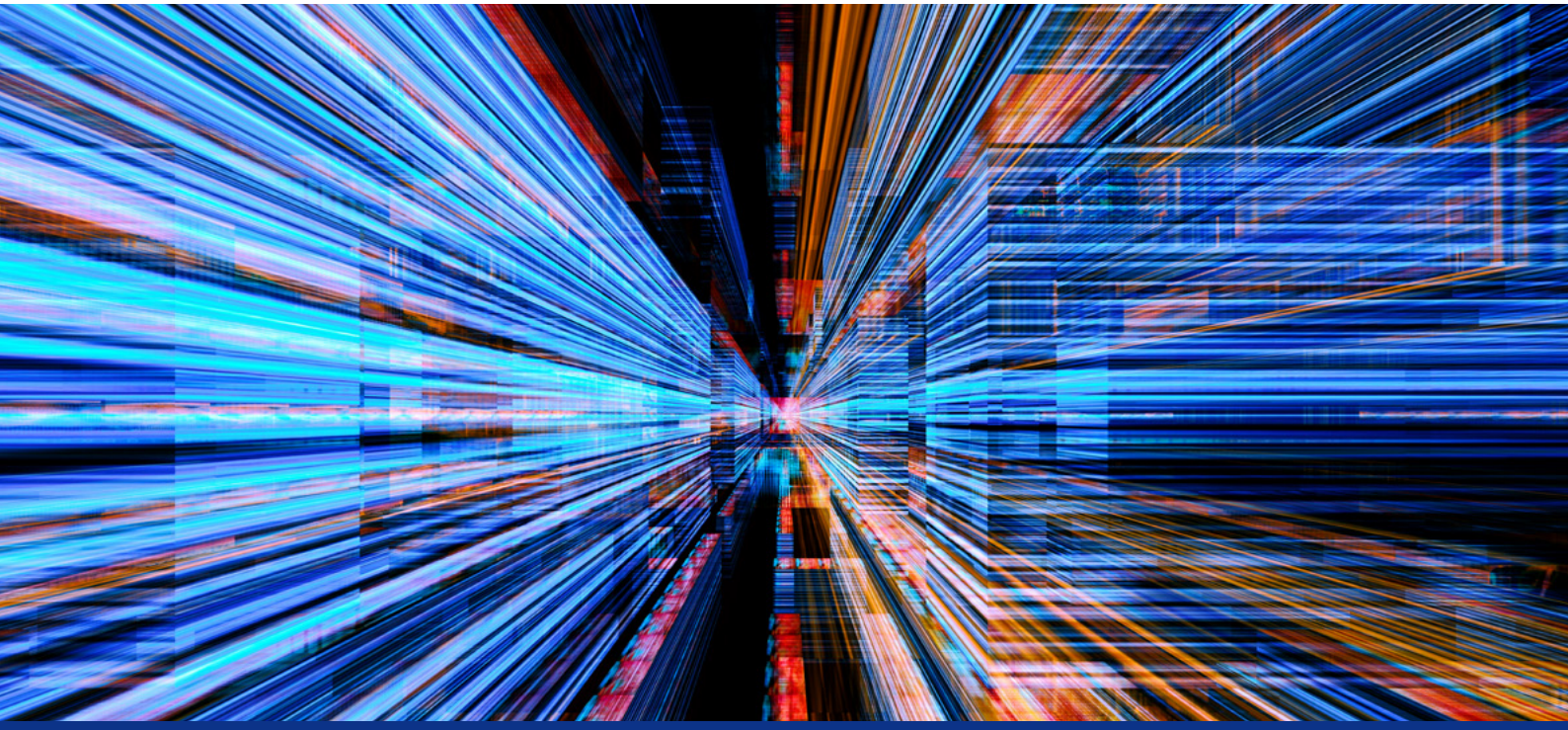
OpenVINO™ ツールキットでコンピューター・ビジョンとディープラーニングを高速化

INTEL CORPORATION

インテルは、ビジョン製品ラインアップの一部として、開発者がエッジからクラウドまで、さまざまなアプリケーションでビジョン・インテリジェンスを利用できるように支援する、新しい OpenVINO™ (Open Visual Inference & Neural Network Optimization) ツールキット (旧インテル® コンピューター・ビジョン SDK) を発表しました。

無料でダウンロード可能な OpenVINO™ ツールキットは、高性能なコンピューター・ビジョンやディープラーニング推論ソリューションの迅速な開発を支援し、異なる種類のインテル® プラットフォームで高速かつ効率的なディープラーニング・ワークロードを提供します ...

この記事の続きはこちら (英語) でご覧になれます。>



MPI-3 の非ブロッキング集合操作による通信レイテンシーの隠蔽

通信と計算のオーバーラップにより HPC パフォーマンスを向上

Amarpal Singh Kapoor インテル コーポレーション コア / ビジュアル・コンピューティング・グループ テクニカル・コンサルティング・エンジニア
Rama Kishan Malladi 同テクニカル・マーケティング・エンジニア
Nitya Hariharan 同テクニカル・マーケティング・エンジニア
Srinivas Sridharan インテル ラボ 並列コンピューティング・ラボ リサーチ・サイエンティスト

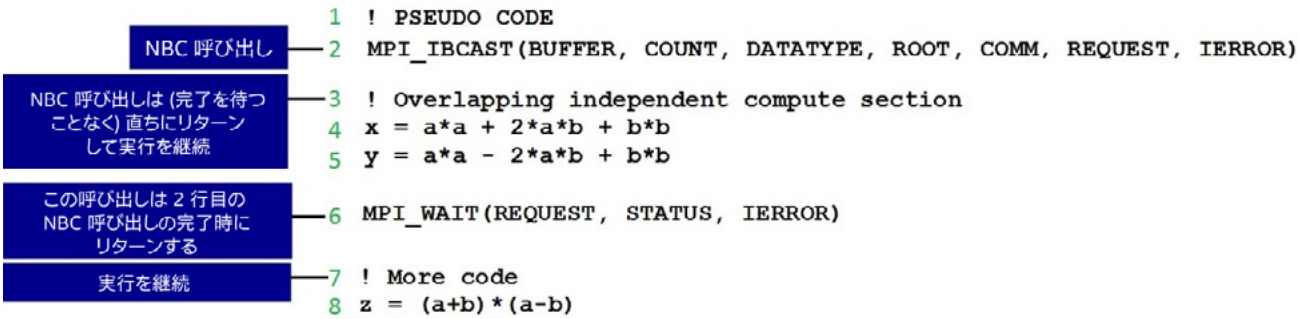
適切に記述された HPC アプリケーションは、多数のノードで直線的に (またはほぼ直線的に) スケーリングします。アプリケーションの直線的なスケーラビリティー / スピードアップは、通信オーバーヘッドの最小化に大きく依存します。

通信オーバーヘッドを最小化する 1 つの方法は、オーバーヘッドを隠蔽することです。非ブロッキング通信関数はメッセージ・パッシング・インターフェイス (MPI) の一部であり、通信と計算をオーバーラップします (つまり、通信レイテンシーを隠蔽します)。MPI の基本的なプロセス間通信 (**`MPI_Send`**、**`MPI_Recv`** など) はブロッキングです (呼び出しは通信が完了するまでリターンしません)。一方、非ブロッキング関数 (**`MPI_Isend`**、**`MPI_Irecv`** など) は直ちにリターンし、通信がバックグラウンドで完了する間に計算を続行できます。

この記事では、MPI-3 標準の比較的新しい非ブロッキング集合 (NBC) 通信関数 (**MPI_Iallreduce**、**MPI_Ibcast** など) について説明します。NBC 関数には多くの長所がありますが、まだ MPI アプリケーションで広く利用されていません。そこで、NBC 関数を分かりやすく説明し、NBC 関数を使用して通信と計算をオーバーラップしてパフォーマンスを向上する方法を示します。

非ブロッキング通信の概要

MPI コードで NBC を使用するということは、単にブロッキング集合関数を置換する (**MPI_Bcast** を非ブロッキング集合関数 **MPI_Ibcast** に置換する) だけではありません。実際に、むやみに変更すると、通信の完了の確認に関連する付加的なオーバーヘッドが発生してパフォーマンスが低下する可能性もあります。**MPI_Bcast** と **MPI_Ibcast** の置換に加えて、コードを慎重に調査し、NBC の処理中に実行できる計算を識別する必要があります。目標は、計算データが通信データに依存しないことを保証することです。保証できない場合、並列化の正当性が損なわれます (**図 1**)。



1 **NBC 通信の例**

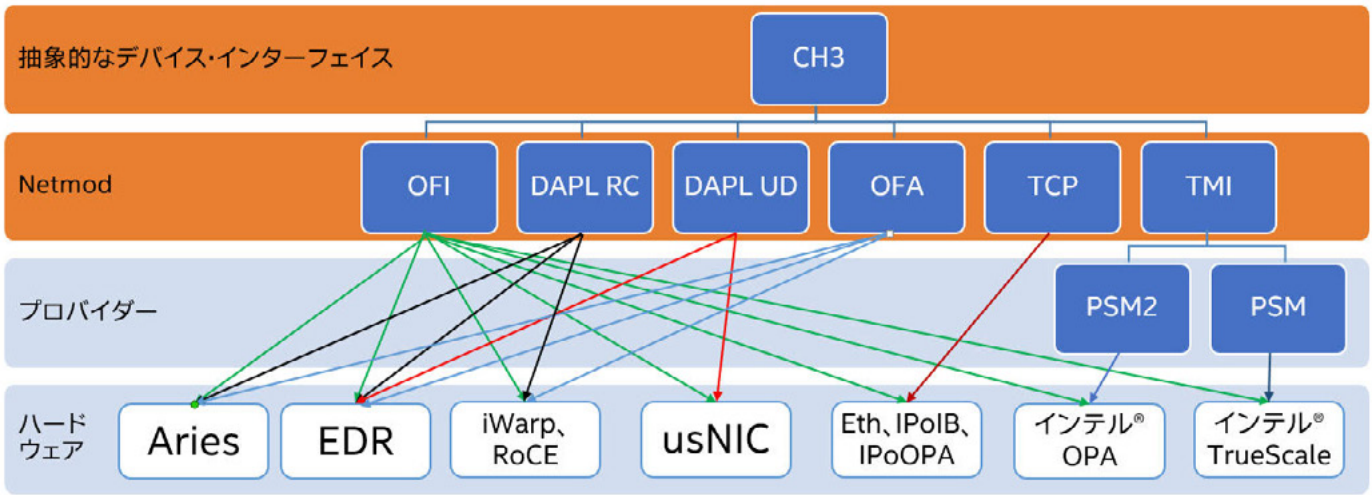
理想的には、オーバーラップしている計算セクションのコードの実行時間を考慮することも重要です。このセクションの実行時間は、非ブロッキング通信によってもたらされる追加の複雑さを調整するため、通信 (この場合は **MPI_Ibcast**) に関連するすべての時間と同じか長くすべきです。

ここで使用されている MPI 実装は、最新の MPI 3.1 標準に完全に準拠した**インテル® MPI ライブラリー 2018 Update 2** です。このライブラリーは、**インテル® MPI ライブラリー単体製品**または**インテル® Parallel Studio XE Cluster Edition** の一部として利用できます。

非ブロッキング通信に関連する課題への対応

ブロッキング通信のセマンティクスは、関連するデータ転送の完了を保証します。しかし、非ブロッキング呼び出しは、完了の確認に **MPI_Wait** や **MPI_Test** のようなヘルパー関数を使用します。非ブロッキングのセマンティクスが完了を保証しないという事実は、計算と通信をオーバーラップする機会を提供します。しかし、ヘルパー関数を使用すると、複数のレベルでオーバーヘッドがカスケードされるため、クリティカル・パスの命令数が増加します。

図 2 にインテル® MPI ライブラリーを構成するソフトウェア・スタックを示します。抽象的なデバイス・インターフェイス・レイヤーの CH3 はインテル® MPI ライブラリーを表しています。ワイヤー (通信手段) に転送するデータを送るには、ハードウェア・レイヤーに到着する前にいくつかのソフトウェア・レイヤー (Netmod およびプロバイダー) を経由する必要があります。



2 **インテル® MPI ライブラリー 2018 ソフトウェア・スタック**

ワイヤーにデータを送る通信呼び出しの起動後に実行する補助命令はすべて、センダーのクリティカル・パスと見なされます。**MPI_Wait** または **MPI_Test** 呼び出し (抽象的なデバイス・インターフェイス・レイヤーから始まる) は、Netmod およびプロバイダー・レイヤーの追加の呼び出しにより完了します。このため、非ブロッキング通信はブロッキング通信よりもコストがかかります。しかし、このコストは、通信の処理中に計算を行うことによるパフォーマンス・ゲインにより相殺されます。

非ブロッキング通信に関連するもう 1 つのオーバーヘッドは、通信を非同期にすることから生じます。非同期処理は通信と計算のオーバーラップを改善しますが、MPI ランクごとに追加のスレッドが必要になります。このスレッドは CPU サイクルを消費するため、理想的には排他的なコアにピニングすべきです。しかし、各ランクで排他的なスレッドピニングを行うと、非ブロッキング MPI 呼び出しの処理を高速化するためだけに、コアの半分を割り当てることになります。そのため、大幅な計算ペナルティーが発生しないように、非同期処理に割り当てるノードあたりのコア数を注意して選択する必要があります。

インテル® MPI ライブラリーでは、次の環境変数を使用して、非同期処理の有効化とコアへのスレッドのピニングを指定します。

```
$ export I_MPI_ASYNC_PROGRESS=1
$ export I_MPI_ASYNC_PROGRESS_PIN=<CPU list>
```

考慮すべきもう 1 つのチューニング・パラメーターは、これらのスレッドの積極性を制御するスピンカウントです。スピンカウントは、次の環境変数を使用して制御します。

```
$ export I_MPI_SPIN_COUNT=<scount>
```

非同期アプローチには、慎重な検討と使用法が必要です。いくつかの代替案は、アイドル状態のコアにスレッドをスケジュールするようにアプリケーションのインバランスを利用します。[1] より洗練された代替案は、ネットワーク・ハードウェアで非同期処理ができるようにすることです。そのようなハードウェア機能が利用可能になれば、非同期処理を使用してパフォーマンスを向上することがはるかに簡単になるでしょう。

格子量子色力学 (QCD)

ここでは、QCD アプリケーションにおける非ブロッキング通信の影響を調べてみましょう。QCD では、格子点 (サイト) に定義されたフェルミオン (クォーク) 場とサイト間のリンクに定義されたゲージ (グルーオン) 場により、時空は 4 次元の超立方体格子上に離散化されます。Wilson-Dslash 演算子は、格子 Dirac Fermion 演算子の変種でゲージ共変微分として使用されます。この演算子は 4 次元の最近傍ステンシル (4D の 9 ポイントステンシル) に似ていて、サイト上のスピノルとリンクに定義されたゲージが複素行列で表現されます。QCD アプリケーションの多くの時間は、共役勾配 (CG) などのスパース反復法ソルバーを使用して、これらの演算子で線形システムを解くことに費やされています。[2]

Wilson-Dslash 演算子の通信パターンは、最近傍の 2 点 (送信と受信) 間の交換です。この計算と通信は、**図 3** で呼び出されている fdopr 関数で実装されます。CG ソルバーコードでは、「赤」と「黒」データを計算するため (赤 - 黒アルゴリズム)、fdopr 関数が 2 回呼び出されています。一般的なマルチノード実装は、Wilson-Dslash 演算子の計算と非ブロッキング MPI 関数 (**`MPI_Isend`**、**`MPI_Irecv`**) を使用した面の通信をオーバーラップします。CG ソルバーコードは、反復法ソルバーの誤差剰余を計算するために集合通信関数 (**`MPI_Allreduce`**) を使用します。

CG ソルバーループの **MPI_Allreduce** 集合操作により、CG ソルバーのパフォーマンスは Wilson-Dslash よりも低くなると予想されます。これは非ブロッキング集合通信を検証する機会です。ここでは、ソルバーのすべての反復で赤の計算の後の集合通信は黒の計算とオーバーラップします (**図 4**)。**MPI_Wait** は非ブロッキング通信の完了を確認するために必要です。(オプションの **MPI_Test** 呼び出しは、保留のリクエストの完了をテストするため、および非ブロッキング通信が開始していない場合に処理を開始するためです。) 非同期通信は関数 **fdopr** 内の計算 (通信) とオーバーラップされます。

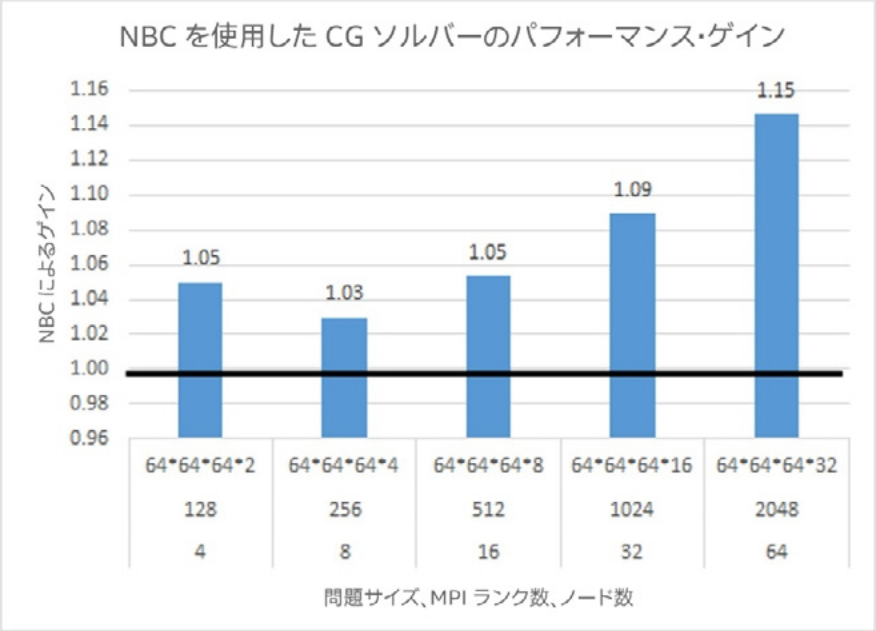
```
      tstart1 = MPI_Wtime()
      call MPI_Allreduce(alphad,ret,1,MPI_DOUBLE_PRECISION,
     +                   MPI_SUM, MPI_COMM_WORLD, ierr)
      call fdopr(atap,p,ap,0,1)
      tend1 = MPI_Wtime()
```

3 **MPI_Allreduce (ブロッキング通信) を使用したベースライン・コード**

```
       tstart1 = MPI_Wtime()
      call MPI_Iallreduce(alphad,ret,1,MPI_DOUBLE_PRECISION,
     +                   MPI_SUM, MPI_COMM_WORLD, ired1, ierr)
      call MPI_Test(ired1, flag1, MPI_STATUS_IGNORE)
      call fdopr(atap,p,ap,0,1)
      call MPI_Wait(ired1,iStatus,ierr)
      tend1 = MPI_Wtime()
```

4 **MPI_Iallreduce (非ブロッキング通信) を使用して最適化されたコード**

NBC を使用して通信と計算をオーバーラップしたことによるパフォーマンスの向上を**図 5** に示します。アプリケーションは、**インテル® Omni-Path ファブリック** (英語) で接続されたデュアル **インテル® Xeon® Gold 6148 プロセッサー**・ノードのクラスターで実行されました。パフォーマンスの向上はノード数が多くなるほど大きくなると予想されますが、**MPI_Allreduce** の時間もノード数とともに増加します。**MPI_Iallreduce** に変更すると、この通信レイテンシーの一部は隠蔽されます。



5 **MPI_Iallreduce によるパフォーマンスの向上**

パフォーマンスの向上

インテル® MPI ライブラリーの NBC は、通信と計算のオーバーラップによりパフォーマンスを向上する可能性を高めます。この最適化に適しているコードセグメントを慎重に検討することが重要です。

参考資料 (英語)

1. Karthikeyan Vaidyanathan, Dhiraj D. Kalamkar, Kiran Pamnany, Jeff R. Hammond, Pavan Balaji, Dipankar Das, Jongsoo Park and Balint Joo. "**Improving Concurrency and Asynchrony in Multithreaded MPI Applications using Software Offloading**," Supercomputing 2015.
2. M.R. Hestenes and E. Stiefel. "**Methods of Conjugate Gradients for Solving Linear Systems**," Journal of Research of the National Bureau of Standards, 1952.

インテル® Parallel Studio XE
純粋なパフォーマンスのためのコードの現代化
無料評価版のダウンロード

intel Software

次のレベルに進もう

hotspot の解析、複数の連続するフレームのキャプチャー、パフォーマンス・ボトルネックの特定—インテル® Graphics Performance Analyzers でゲームのパフォーマンスを最大限に引き出しましょう。

無料のダウンロード (英語) >

コンパイラーの最適化に関する詳細は、最適化に関する注意事項 (software.intel.com/en-us/articles/optimization-notice#opt-jp) を参照してください。
Intel、インテル、Intel ロゴは、アメリカ合衆国および / またはその他の国における Intel Corporation の商標です。
© Intel Corporation.

Software

THE PARALLEL UNIVERSE

ベンチマーク結果は、「Spectre」および「Meltdown」と呼ばれる脆弱性への対処を目的とした最新のソフトウェア・パッチおよびファームウェア・アップデートの適用前に取得されたものです。パッチやアップデートを適用したデバイスやシステムでは同様の結果が得られないことがあります。

インテル® コンパイラーでは、インテル® マイクロプロセッサーに限定されない最適化に関して、他社製マイクロプロセッサー用に同等の最適化を行えないことがあります。これには、インテル® ストリーミング SIMD 拡張命令 2、インテル® ストリーミング SIMD 拡張命令 3、インテル® ストリーミング SIMD 拡張命令 3 補足命令などの最適化が該当します。インテルは、他社製マイクロプロセッサーに関して、いかなる最適化の利用、機能、または効果も保証いたしません。本製品のマイクロプロセッサー依存の最適化は、インテル® マイクロプロセッサーでの使用を前提としています。インテル® マイクロアーキテクチャーに限定されない最適化のなかにも、インテル® マイクロプロセッサー用のものがあります。この注意事項で言及した命令セットの詳細については、該当する製品のユーザー・リファレンス・ガイドを参照してください。注意事項の改訂 #20110804

インテル® テクノロジーの機能と利点はシステム構成によって異なり、対応するハードウェアやソフトウェア、またはサービスの有効化が必要となる場合があります。

実際の性能はシステム構成によって異なります。絶対的なセキュリティーを提供できるコンピューター・システムはありません。詳細については、各システムメーカーまたは販売店にお問い合わせいただくか、http://www.intel.co.jp/ を参照してください。

性能に関するテストに使用されるソフトウェアとワークロードは、性能がインテル® マイクロプロセッサー用に最適化されていることがあります。SYSmark* や MobileMark* などの性能テストは、特定のコンピューター・システム、コンポーネント、ソフトウェア、操作、機能に基づいて行ったものです。結果はこれらの要因によって異なります。製品の購入を検討される場合は、他の製品と組み合わせた場合の本製品の性能など、ほかの情報や性能テストも参考にして、パフォーマンスを総合的に評価することをお勧めします。詳細については、www.intel.com/performance（英語）を参照してください。

インテルは、本資料で参照しているサードパーティーのベンチマーク・データまたは Web サイトの設計や実装について管理や監査を行っていません。本資料で参照している Web サイトまたは類似の性能ベンチマーク・データが報告されているほかの Web サイトも参照して、本資料で参照しているベンチマーク・データが購入可能なシステムの性能を正確に表しているかを確認されるようお勧めします。

この文書および情報は、インテルのお客様向けの参考情報として記載されているものであり、現状のまま提供され、明示されているか否かにかかわらず、いかなる保証もいたしません。ここにいう保証には、商品適格性、特定目的への適合性、知的財産権の非侵害性への保証を含みますが、これらに限定されるものではありません。本資料は、本資料に記述、表示、または記載されたいかなる知的財産権のライセンスも許諾するものではありません。インテル製品は、医療、救命、延命措置、重要な制御または安全システム、核施設などの目的に使用することを前提としたものではありません。

© 2018 Intel Corporation. 無断での引用、転載を禁じます。Intel、インテル、Intel ロゴ、Intel Core、Xeon、OpenVINO、VTune は、アメリカ合衆国および / またはその他の国における Intel Corporation の商標です。

* その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

JPN/1808/PDF/XL/SSG/SS