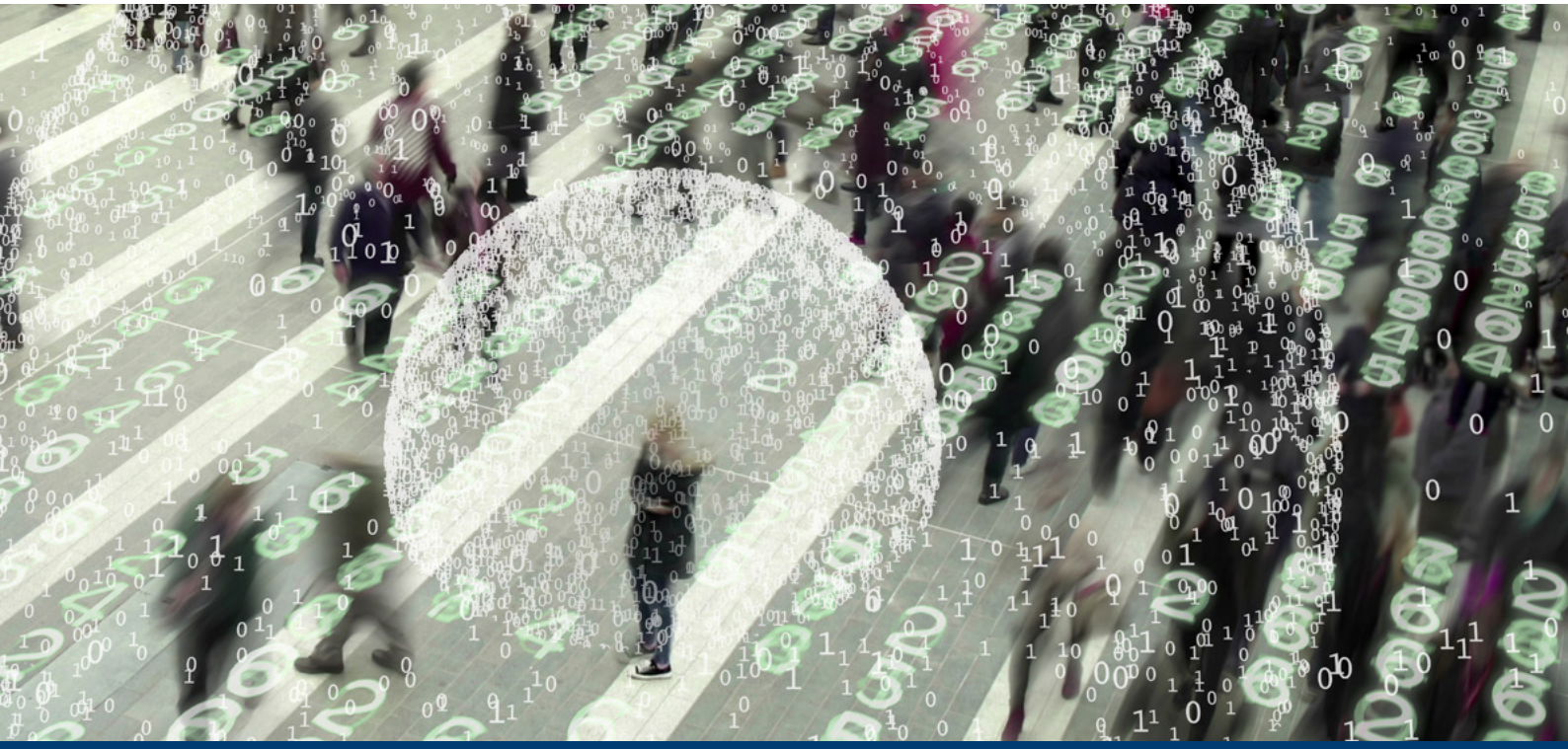


ビッグデータ解析とマシンラーニングの有効利用

インテル® パフォーマンス・ライブラリーにより実現

Vadim Pirogov 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
Ivan Kuzmin 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
Sarah Knepper インテル コーポレーション ソフトウェア開発エンジニア

現代の社会では、天気予報から救命薬の発見に至る広範な分野において、さまざまなエンジニアリングの問題を解決するため、コンピューターへの依存度が高まっています。私たちは今、意思決定や複雑な問題を解くことにおいて、マシンが人々と同等またはそれ以上の能力を発揮するという、劇的な変化を目の当たりにしつつあります。すでに、**Jeopardy*** や**囲碁**では、コンピューターがトッププレーヤーに勝利を収め、自動運転車がカリフォルニア州の道路を走行しています。これらはすべて、(ムーアの法則による) ペタフロップ・レベルの計算能力と、マシンラーニング・アルゴリズムのトレーニングに利用可能な膨大な量のデータにより実現されます。

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。

インテルは、今後のマルチコア / メニーコア計算プラットフォームのハードウェアおよびソフトウェア・アーキテクチャーの課題に取り組むため、最先端の学術研究者および業界関係者と緊密に協力しています。また、マシンラーニングの複雑さに対応できるように、インテル® Data Analytics Acceleration Library (インテル® DAAL) やインテル® マス・カーネル・ライブラリー (**インテル® MKL**) などのインテル® ソフトウェア・ツールを利用して開発者がパフォーマンスの最適化を行えるように支援しています。

マシンラーニングの課題

この 10 年間、マシンラーニングは急激に成長しました。この成長は、膨大な量のデータを生成するインターネットによって後押しされました。データからパターンを抽出し、その情報を適用して予測するため、新しいアプローチとアルゴリズムが開発されました。そして、計算能力の指数関数的成長により、それらのアルゴリズムを膨大なデータセットに適用して、有用な予測を行うことが可能になりました。

ディープ・ニューラル・ネットワーク (DNN) は、マシンラーニング分野の最先端のアルゴリズムです。1990 年代後半に業界での採用が進み、当初は銀行小切手の手書き文字認識などのタスクに利用されていました。ディープ・ニューラル・ネットワークは、このタスクにおいて人と同等またはそれ以上の能力を提供します。今日 DNN は、画像認識、ビデオおよび自然言語処理、自動運転などに必要な複雑な視覚理解の問題の解決に使用されています。DNN には、非常に多くの計算リソースと処理データが必要です。一例として、最新の画像認識トポロジー AlexNet のトレーニングには最新の計算システムで数日を要し、1400 万を上回る画像を使用します。この複雑さに対応するには、トレーニング時間を短縮し、産業分野のニーズに応えることができる高度に最適化されたビルディング・ブロックが必要です。

インテル® MKL

インテル® MKL は、インテル® アーキテクチャーおよび互換アーキテクチャー向けのハイパフォーマンス数学ライブラリーです (**図 1**)。現在および将来のインテル® プロセッサー向けに最適化された一般的な密 (BLAS、LAPACK) およびスパース (スパース BLAS、インテル® MKL PARDISO) 線形代数ルーチン、離散フーリエ変換、ベクトル演算、統計関数の実装を提供します。インテル® MKL は、命令 / スレッド / クラスターレベルの並列性を利用して、ワークステーション、サーバー、スーパーコンピューター上でさまざまな科学、工学、金融アプリケーションのパフォーマンスを向上します。

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。

より多くのコア → より多くのスレッド → より幅広いベクトル

	インテル® Xeon® プロセッサー 64 ビット	インテル® Xeon® プロセッサー 5100 製品ファミリー	インテル® Xeon® プロセッサー 5500 製品ファミリー	インテル® Xeon® プロセッサー 5600 製品ファミリー	インテル® Xeon® プロセッサー E5-2600 V2 製品ファミリー	インテル® Xeon® プロセッサー E5-2600 V3 製品ファミリー V4 製品ファミリー	将来のインテル® Xeon® プロセッサー[1]	インテル® Xeon Phi™ コプロセッサー x100 製品ファミリー (開発コード名 Knights Corner)	インテル® Xeon Phi™ プロセッサー/コプロセッサー x200 製品ファミリー (開発コード名 Knights Landing)	将来のインテル® Xeon Phi™ 製品ファミリー (開発コード名 Knights Hill)
最大コア数	1	2	4	6	12	18-22	*TBD*	61	72	*TBD*
最大スレッド数	2	2	8	12	24	36-44	*TBD*	244	288	*TBD*
SIMD 幅	128	128	128	128	256	256	512	512	512	*TBD*
ベクトル ISA	インテル® SSE3	インテル® SSE3	インテル® SSE4-4.1	インテル® SSE4.2	インテル® AVX	インテル® AVX2	インテル® AVX-512	インテル® IMCI	インテル® AVX-512	*TBD*

リリース済みの製品の仕様は、ark.intel.com (英語) を参照してください。インテル製品は予告なく変更されることがあります。本資料に記載されているすべての日付および製品は、計画以外の目的ではご利用になれません。1. 未リリースまたは計画段階

1 インテル® MKL のターゲット・プロセッサー

インテル® MKL は、ハイパフォーマンス・コンピューティング (HPC) 向けに設計されていますが、その機能は数学と同様にあらゆるものに利用できます。行列 - 行列乗算、高速フーリエ変換 (FFT)、ガウス消去法のような関数は、数学や工学関連の多くの問題の基礎となるだけでなく、マシンラーニング・アルゴリズムの基礎にもなります。

インテル® MKL 2017 には、主なマシンラーニング・アルゴリズムに利点をもたらす最適化された機能に加えて、マシンラーニングに固有の計算ニーズに対応する新しい DNN 拡張が含まれています。ここでは、ディープ・ニューラル・ネットワークの拡張と行列 - 行列乗算の改善点について説明します。

インテル® MKL による DNN の高速化

インテル® MKL の DNN プリミティブ

ディープラーニングは、複数の処理層から成るグラフ構造を使用して、ハイレベルの抽象化をモデル化するマシンラーニングの一種で、データセンターで急速に採用が進んでいます。このアプローチは、生物が視覚野を通して現実を認識する方法からアイデアを得て開発されました。画像 / ビデオ認識、自然言語処理、レコメンダー・システムなどのアプリケーションで幅広く活用されています。これらのワークロードは、多数のアルゴリズムを利用しており、特に多次元畳み込みと行列 - 行列乗算に依存しています。畳み込みは行列乗算として表現することができますが、場合によっては、直接畳み込みとして実装したほうが、現代のアーキテクチャーで大幅なパフォーマンスの向上を達成できる可能性があります。

畳み込みに加えて、ディープラーニングのワークロードには、小さな次元の行列を扱う数種類の層が含まれます。データ変換のオーバーヘッドを最小限に抑えるため、インテル® MKL 2017 の新しいディープ・ニューラル・ネットワーク (DNN) ドメインでは、主要関数の最適化された実装が追加されています。

DNN ドメインには、AlexNet、VGG、GoogLeNet、ResNet を含む主要画像認識トポロジーの高速化に必要な関数が含まれます。

これらの DNN トポロジーは、テンソルと呼ばれる多次元データを扱う、多くの標準のビルディング・ブロックやプリミティブに依存します。プリミティブには、畳み込み、正規化、アクティベーション、および内積に加えて、テンソルの操作に必要な関数が含まれます。インテル® アーキテクチャー上で計算を効率良く実行するには、ベクトル化により SIMD 命令を利用したり、スレッド化により複数の計算コアを利用する必要があります。最近のプロセッサーは、最大 512 ビットのベクトルデータ (16 の単精度数) を処理でき、1 サイクルごとに最大 2 つの積和演算 (FMA: Fused Multiply-Add) 命令を実行できるため、ベクトル化は非常に重要です。ベクトル化による利点を得るためには、データを連続するメモリー位置に配置する必要があります。通常、テンソルの次元は小さいため、データレイアウトの変更により大きなオーバーヘッドが生じます。そのため、プリミティブ間でデータレイアウトを変更せずに、トポロジーのすべての操作を実行するようにします。

インテル® MKL は、よく使用される操作をベクトル化に対応したデータレイアウトで実装したプリミティブを提供します。

- 直接バッチ畳み込み
- 内積
- プーリング : 最大、最小、平均
- 正規化 : チャネル、バッチ正規化にわたる局所反応正規化 (LRN)
- アクティベーション : 正規化線形関数 (ReLU)
- データ操作 : 多次元転置 (変換)、分割、連結、合計、スケール

BLOG HIGHLIGHTS

マシンに教えられたのですか?!

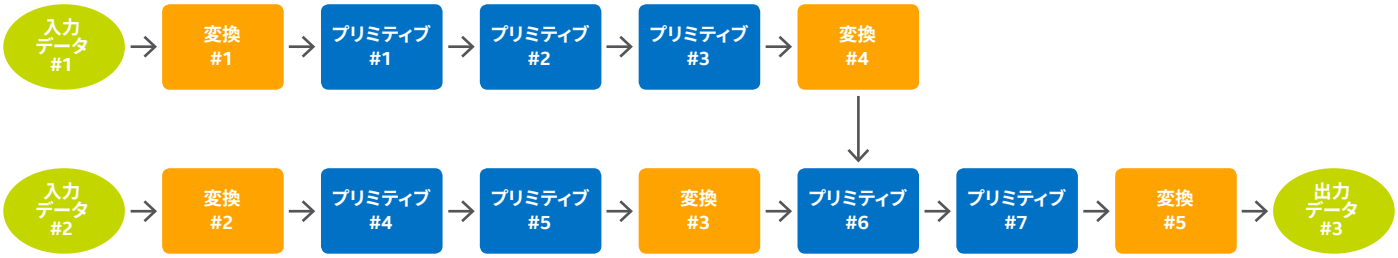
LENNY TRAN >

データ・サイエンティスト、開発者、研究者は、マシンラーニングを利用して、これまで得ることができなかった洞察を得ています。経験から学ぶプログラムをヒトゲノムの解明に役立てたり、消費者行動の前後関係や背景を理解するプログラムによりリアルタイムでお勧めを作成したり、プログラムが画像や人の顔を認識/識別できるようにしてセキュリティーを向上することができます。

この記事の続きはこちら (英語) でご覧になれます。 >

ニューラル・ネットワーク・トポロジーの実行フローは、セットアップと実行の 2 つのフェーズで構成されます。セットアップ・フェーズでは、スコアリング、トレーニング、その他のアプリケーション固有の計算を実装するのに必要なすべての DNN 操作の説明をアプリケーションが作成します。1 つの DNN 操作から次の DNN 操作へデータを渡すため、適切な出力と入力データレイアウトが一致しない場合、アプリケーションは中間データを変換し、一時配列を割り当てることがあります。このフェーズは、通常のアプリケーションでは一度だけ実行され、その後、複数の実行フェーズで実際の計算が行われます。

実行フェーズ (**図 2**) では、データは NCWH (バッチ、チャネル、幅、高さ) などのプレーンなレイアウトでネットワークに渡され、SIMD 対応レイアウトに変換されます。層と層の間のデータの伝達では、データレイアウトが保持され、既存の実装でサポートされていない操作を実行するのに必要な場合は変換されます。



2 入力ステップ

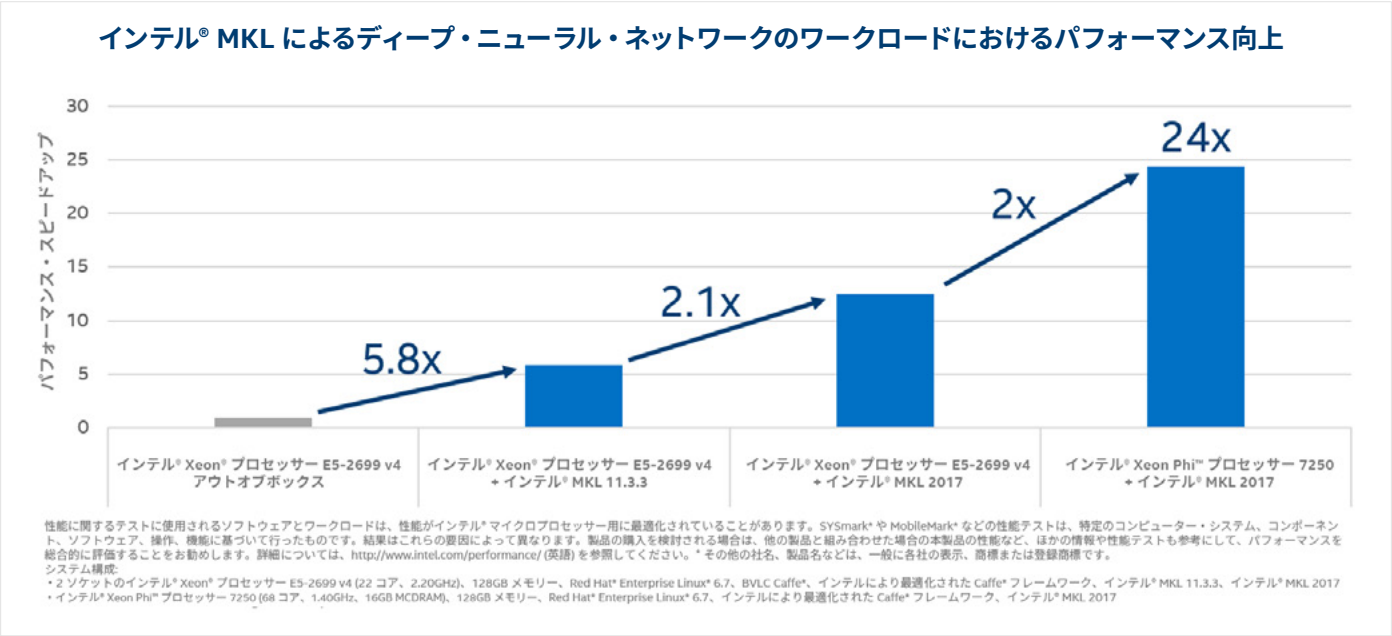
ケーススタディー : Caffe*

Caffe* は、Berkeley Vision and Learning Center (BVLC) によって開発されたディープラーニング・フレームワークで、最もよく使用されている画像認識コニュニティー・フレームワークの 1 つです。画像認識ニューラル・ネットワーク・トポロジーの **AlexNet** とラベル付き画像データベースの **ImageNet** とともに、Caffe* はベンチマークとしてよく使用されます。

Caffe* は、標準の BLAS インターフェイスでインテル® MKL の最適化された数学ルーチンを利用しています。しかし、オリジナルの実装では、効率良く並列化されず、畳み込みの実装で GEMM 関数の利点を最大限に引き出すことができませんでした。GEMM をより大きな行列で実行するように計算フローを再構築し、OpenMP* を利用してさらなる並列化を行ったところ、2 ソケットのインテル® Xeon® プロセッサー E5-2699 v4 ベースのシステムで、AlexNet トポロジーのトレーニングが 5.8 倍スピードアップしました。前述のように、直接畳み込みとして実装したほうが効率的です。新しいインテル® MKL 2017 のプリミティブを使用することで、11 倍のスピードアップを達成できました。

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。

インテル® アーキテクチャーとインテル® MKL は進化しています。2016 年 6 月、最大 72 コアと高速なオンチップメモリーを搭載した、インテル® アドバンスト・ベクトル・エクステンション 512 (インテル® AVX-512) 対応の超並列アーキテクチャーである、インテル® Xeon Phi™ プロセッサー x200 製品ファミリーがリリースされました。新しいインテル® MKL プリミティブに対応した**インテルの Caffe*** は、1 ソケットのインテル® Xeon Phi™ プロセッサー 7200 でさらに 2 倍のスピードアップをもたらしました (**図 3**)。



3 Caffe*/AlexNet シングルノード・トレーニング・パフォーマンス

高速な行列 - 行列乗算

行列 - 行列乗算 (GEMM) は、多くの科学、工学、マシンラーニング・アプリケーションで利用される基本操作です。この操作の最適化は継続的に求められています。インテル® MKL は、ハイパフォーマンスな並列 GEMM 実装を提供します。最適なパフォーマンスを提供するため、インテル® MKL の GEMM 実装は通常、オリジナルの入力行列をターゲット・プラットフォームに最適な内部データ形式に変換します。このデータ変換 (パッキングと呼ばれる) にはコストがかかり、特に 1 つ以上の小さな次元を含む入力行列ではコストが大きくなります。インテル® MKL 2017 では、冗長なパッキングを回避するため、[S,D]GEMM パックド・アプリケーション・プログラム・インターフェイス (API) が追加されています。この API は、一部の行列サイズでパフォーマンスを向上し、マシンラーニングでも利用できます。

この API を利用して、行列を明示的に内部パックド形式に変換し、(1 つまたは複数の) パックド行列を複数の GEMM 呼び出しに渡します。このアプローチでは、反復性ニューラル・ネットワークのように入力行列 (A または B) が複数の GEMM 呼び出しで再利用される場合、パッキングコストを相殺できます。

例

次の 3 つの GEMM 呼び出しは、同じ行列 A を使用しますが、行列 B と C は呼び出しごとに異なります。

```
float *A, *B1, *B2, *B3, *C1, *C2, *C3, alpha, beta;
MKL_INT m, n, k, lda, ldb, ldc;
// ポインターと行列の次元の初期化 (スキップ)
sgemm("T", "N", &m, &n, &k, &alpha, A, &lda, B1, &ldb, &beta, C1, &ldc);
sgemm("T", "N", &m, &n, &k, &alpha, A, &lda, B2, &ldb, &beta, C2, &ldc);
sgemm("T", "N", &m, &n, &k, &alpha, A, &lda, B3, &ldb, &beta, C3, &ldc);
```

この場合、行列 A は各 sgemm 呼び出しで内部パックドデータ形式に変換されます。行列 A を 3 回パッキングする相対コストは、n (行列 B と C の列数) が小さい場合、高くなります。次のように、行列 A のパッキングを 1 回にし、パックド形式を 3 つの連続する GEMM 呼び出しで使用することで、このコストを最小限に抑えることができます。

```
// パックドデータ形式のメモリー割り当て
float *Ap;
Ap = sgemm_alloc("A", &m, &n, &k);
// A をパックド形式に変換
sgemm_pack("A", "T", &m, &n, &k, &alpha, A, &lda, Ap);
// 行列 A のパックド形式 Ap を使用して SGEMM 計算を実行
sgemm_compute("P", "N", &m, &n, &k, Ap, &lda, B1, &ldb, &beta, C1, &ldc);
sgemm_compute("P", "N", &m, &n, &k, Ap, &lda, B2, &ldb, &beta, C2, &ldc);
sgemm_compute("P", "N", &m, &n, &k, Ap, &lda, B3, &ldb, &beta, C3, &ldc);
// Ap のメモリーを解放
sgemm_free(Ap);
```

BLOG HIGHLIGHTS

Linux* 誕生から 25 年: オープンソース革命のリーダー

IMAD SOUSOU >

1991 年 8 月 25 日にあなたはどこにいましたか? 私は自分がどこで何をしていたか覚えていませんが、その日起こったことは私の人生とキャリアに大きな影響を与えることになりました。25 年前のこの日に Linus Torvalds 氏が Linux* オペレーティング・システムを発表したことで、1991 年 8 月 25 日はテクノロジーの歴史において最も重要な日の 1 つになりました。

インテルは、長年にわたり Linux* と良好な関係を築いてきました。市場でのリーダーシップ、優れたパフォーマンス、豊富なドキュメントが、インテル® アーキテクチャー (IA) での Linux* の採用につながりました。そして、386 から今日の強力なコンピューティング・システムへと、Linux* はインテル® プラットフォーム上で成長を遂げました。インテルは、ハードウェアだけでなく、1990 年代半ばから Linux* ソフトウェア・コミュニティーにおいても積極的に貢献してきました。インテルの Linux* に関する最初の画期的な取り組みは、1999 年にリリースされたインテル® Dot. Station です。これは、インテル初の Linux* ベースのデバイスであるだけでなく、インテル初のコンシューマー向けオペレーティング・システムでもあります。

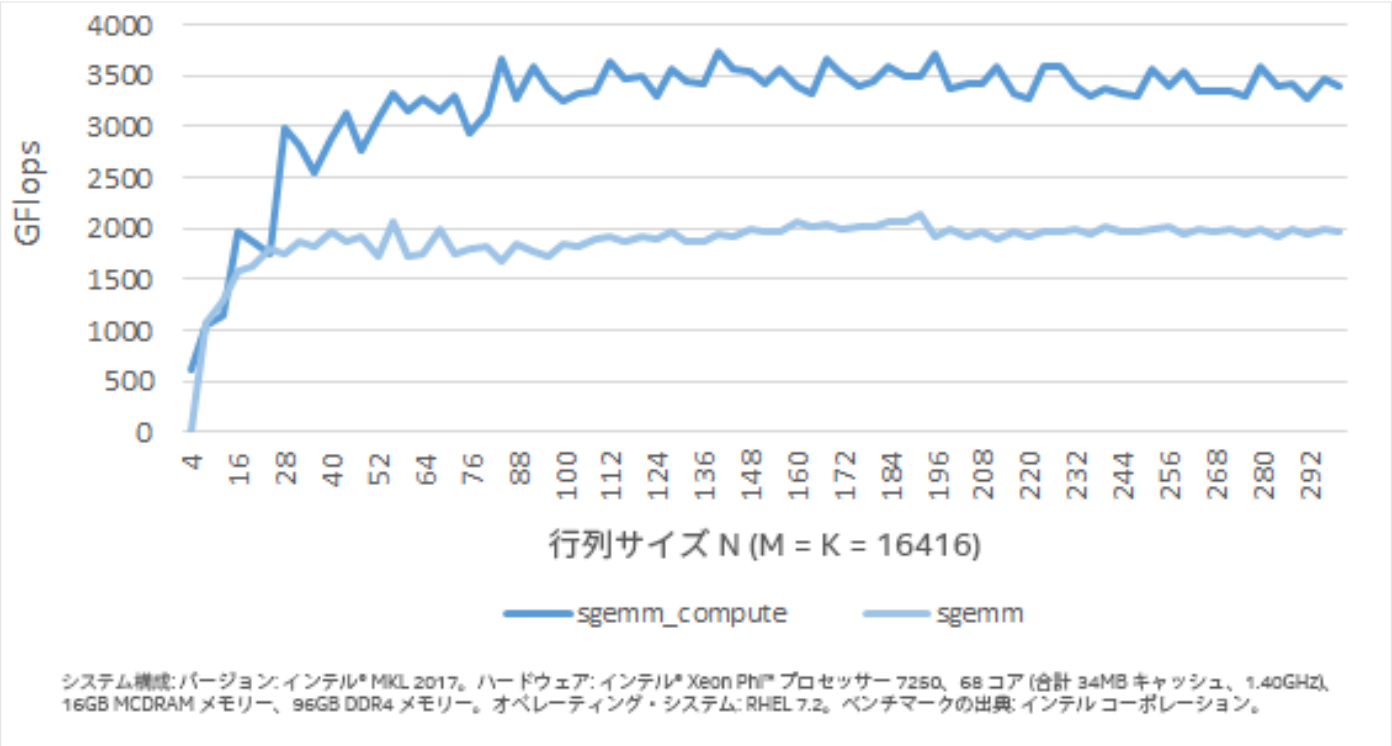
この記事の続きはこちら (英語) でご覧になれます。>

このサンプルコードでは、GEMM パックド API をサポートするために追加された 4 つの新しい関数を使用しています : sgemm_alloc、sgemm_pack、sgemm_compute、および sgemm_free。最初に、sgemm_alloc を使用して、パックド形式に必要なメモリーを割り当てます。sgemm_alloc は、パックド行列を識別する文字引数 (ここでは A) と行列の次元を示す 3 つの整数引数を受け付けます。次に、sgemm_pack がオリジナルの A 行列をパックド形式 Ap に変換し、アルファ・スケーリングを実行します。オリジナルの行列 A は変更されません。3 つの sgemm 呼び出しは、パックド行列を扱い、alpha=1.0 を仮定する 3 つの sgemm_compute 呼び出しに置換されています。sgemm_compute の最初の 2 つの文字引数は、行列 A がパックド形式 (“P”) で、行列 B が 転置されていない列優先形式 (“N”) であることを示します。最後に、sgemm_free を呼び出して、Ap に割り当てられたメモリーを解放します。

GEMM パックド API は、3 つの行列 - 行列乗算用に行列 A を 2 回パッキングするコストを排除します。複数の GEMM 呼び出しで入力行列 A や B が再利用される場合、パックド API を利用することで A や B のデータ変換コストを排除できます。

パフォーマンス

図 4 は、インテル® Xeon Phi™ プロセッサー 7250 上でパックド API を利用した場合のパフォーマンス・ゲインを示します。パッキングコストは、同じ行列 A を使用する多数の SGEMM 呼び出しにより完全に相殺できると仮定されます。比較のため、通常の SGEMM 呼び出しのパフォーマンスも掲載されています。



4 インテル® Xeon Phi™ プロセッサー (68 スレッド) 上でのパックド行列 A に対する sgemm_compute のパフォーマンス

実装の注意点

最高のパフォーマンスを達成するためには、gemm_pack と gemm_compute の呼び出しで同じスレッド数を使用することを推奨します。同じ行列 A または B を共有する GEMM 呼び出しが少ない場合、パックド API によってもたらされるパフォーマンスの向上はわずかです。

gemm_alloc ルーチンは、オリジナルの入力行列とほぼ同じサイズのメモリーを割り当てます。つまり、大きな入力行列では、非常に多くのメモリーが必要になります。

GEMM パックド API は、インテル® MKL 2017 では SGEMM と DGEMM でのみ実装されています。すべてのインテル® アーキテクチャーで利用できますが、64 ビットのインテル® AVX2 以上向けにのみ最適化されています。

インテル® DAAL

マシンラーニングとディープラーニングのアルゴリズムが広く使用されるようになった主な要因は、アルゴリズムのトレーニングに利用可能なデータの急増です。「ビッグデータ」という用語にはいくつかの定義があります。ここでは、多様性、速度、量という ビッグデータの 3 つの特性について説明します。これらの特性は、それぞれ特別な計算ソリューションを必要とし、インテル® DAAL はそのすべてに対応するように設計されています。

多様性

「多様性」とは、データ抽出や解析において特定の課題が生じる、構造化 / 非構造化データの多くのソースと種類を指します。ビッグデータのこの特性に対応するため、インテル® DAAL ではライブラリーの主要部分としてデータ管理コンポーネントを提供しています。このコンポーネントには、データの取得、前処理と正規化、サポートされるデータソースから数値形式へのデータ変換、モデル表現用のクラスとユーティリティーが含まれます。実際の解析アプリケーションでは、どの部分でもボトルネックが見つかる可能性があるため、インテル® DAAL はデータフロー全体を最適化するため、データ解析のすべてのステージをカバーする最適化されたビルディング・ブロックを提供します : データソースからのデータの取得、前処理、変換、データマイニング、モデリング、検証、および意思決定 **(図 5)**。

5 データ解析のステージ

さらに、ホモジニアスとヘテロジニアスの両方のデータをサポートし、必要に応じて、中間データの変換を行います。密 / スパースデータ型をサポートし、ノイズの多いデータを扱うことができるアルゴリズムも提供します。

速度

今日のビジネス環境において計算速度は重要です。ビッグデータを扱う際に最も重要なことの 1 つは、データをリアルタイムに受け取り、素早く応答することです。インテル® DAAL は、アプリケーションの開発期間を短縮し、パフォーマンスの向上を支援するように設計されています。アプリケーションがより早く、より優れた予測を行い、利用可能な計算リソースに応じてスケーリングできるようにします。インテル® DAAL はインテル® MKL を利用して、インテル® アーキテクチャー上でパフォーマンスを最大限引き出せるように、追加のアルゴリズムの最適化を提供します。C++、Java*、および Python* 向けの API が用意されているため、モデルのプロトタイプを素早く作成し、大規模クラスター環境へ簡単に配置できます。

量

ビッグデータとは、膨大な量のデータを意味します。メモリーに収まらない可能性もあります。また、クラスターの異なるノードや複数のデバイス上にデータセットが分散していることも珍しくありません。このような状況に対応するため、インテル® DAAL のアルゴリズムは、バッチ、オンライン、分散処理計算モードをサポートしています。

バッチ処理モードでは、アルゴリズムはデータセット全体を処理して最終結果を生成します。処理を行う時点でデータセット全体が利用可能でないか、データセットがデバイスメモリーに収まらない場合は、ほかの 2 つの処理モードを使用する必要があります。

オンライン処理モードでは、アルゴリズムはデバイスのメモリーにストリームされるブロック単位でデータを処理し、部分結果を段階的に更新して、最後のデータブロックを処理したらファイナライズします。

分散処理モードでは、アルゴリズムは複数のデバイス (計算ノード) に分散されたデータを処理します。各ノードで部分結果を生成し、最終的にマスターノードですべての部分結果をマージします。

分散処理に利用可能なプラットフォームは多数あるため、インテル® DAAL の分散アルゴリズムは、デバイス間の通信技術に依存しないように抽象化されています。そのため、インテル® DAAL は、**MPI**、Hadoop*、Spark* ベースの環境を含む、さまざまなマルチデバイス処理およびデータ転送シナリオで利用できます。ユーザーがデバイス間の通信を実装する必要がありますが、インテル® DAAL では、一般的な解析プラットフォームでの使用例を示すサンプルが用意されています。

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。

異なる処理モードでのインテル® DAAL の使用

バッチ処理モード

インテル® DAAL のすべてのアルゴリズムは、バッチ処理モードをサポートしており、比較的簡単に使用することができます。バッチ処理モードでは、特定のアルゴリズム・クラスの compute メソッドのみ使用されます。使用するアルゴリズムを選択し、入力データとアルゴリズムのパラメーターを設定します。そして、compute メソッドを実行し、結果にアクセスします。次の例は、コレスキー分解アルゴリズムでインテル® DAAL をバッチ処理モードで使用する方法を示します。

```
// デフォルトのメソッドでコレスキー分解を計算するアルゴリズムを作成
cholesky::Batch<> cholesky_alg;

// 入力データを設定
cholesky_alg.input.set(cholesky::data, dataSource.getNumericTable())
// compute を実行
cholesky_alg.compute();

// 計算結果にアクセス
services::SharedPtr<cholesky::Result> res = cholesky_alg.getResult()
```

オンライン処理モード

インテル® DAAL の一部のアルゴリズムは、データセットをブロック単位で処理することができます。オンライン処理モードの API は、バッチ処理モードと似ていますが、データが利用可能になるたびに compute メソッドを実行し、最後に **finalizeCompute** を呼び出して結果を組み合わせる点が異なります。次の例は、特異値分解 (SVD) アルゴリズムでインテル® DAAL をオンライン処理モードで使用する方法を示します。

```
// SVD アルゴリズムを初期化
svd::Online<double, defaultDense> algorithm;

// ループ内のデータブロックを処理
Status loadStatus;
while((loadStatus = dataSource.loadDataBlock(nRowsInBlock)) == success)
{
      // 入力データを設定
      algorithm.input.set( svd::data, dataSource.getNumericTable() );
//  compute を実行
      algorithm.compute();
}

// 計算をファイナライズして SVD 結果を取得
algorithm.finalizeCompute();
SharedPtr<svd::Result> res = algorithm.getResult();

// 結果にアクセス
printNumericTable(res->get(svd::singularValues), "Singular values:");
```

分散処理モード

インテル® DAAL の一部のアルゴリズムは、複数のデバイスに分散されたデータセットを処理できます。アルゴリズムに応じて、異なる計算スキームを適用する必要があります。適用するスキームは、単純な MapReduce (最初にローカルノードで実行し、最後にマスターノードで実行する) から MapReduce-Map のような複雑なスキームまでさまざまです。インテル® DAAL のプログラミング・ガイドには、インテル® DAAL を Hadoop*、Spark*、MPI ベースの環境で使用する方法を示すサンプルが含まれています。次の例は、主成分分析 (PCA) アルゴリズムでインテル® DAAL を分散処理モードで使用する方法を示します。

ステップ 1: ローカルノード (マッパー)

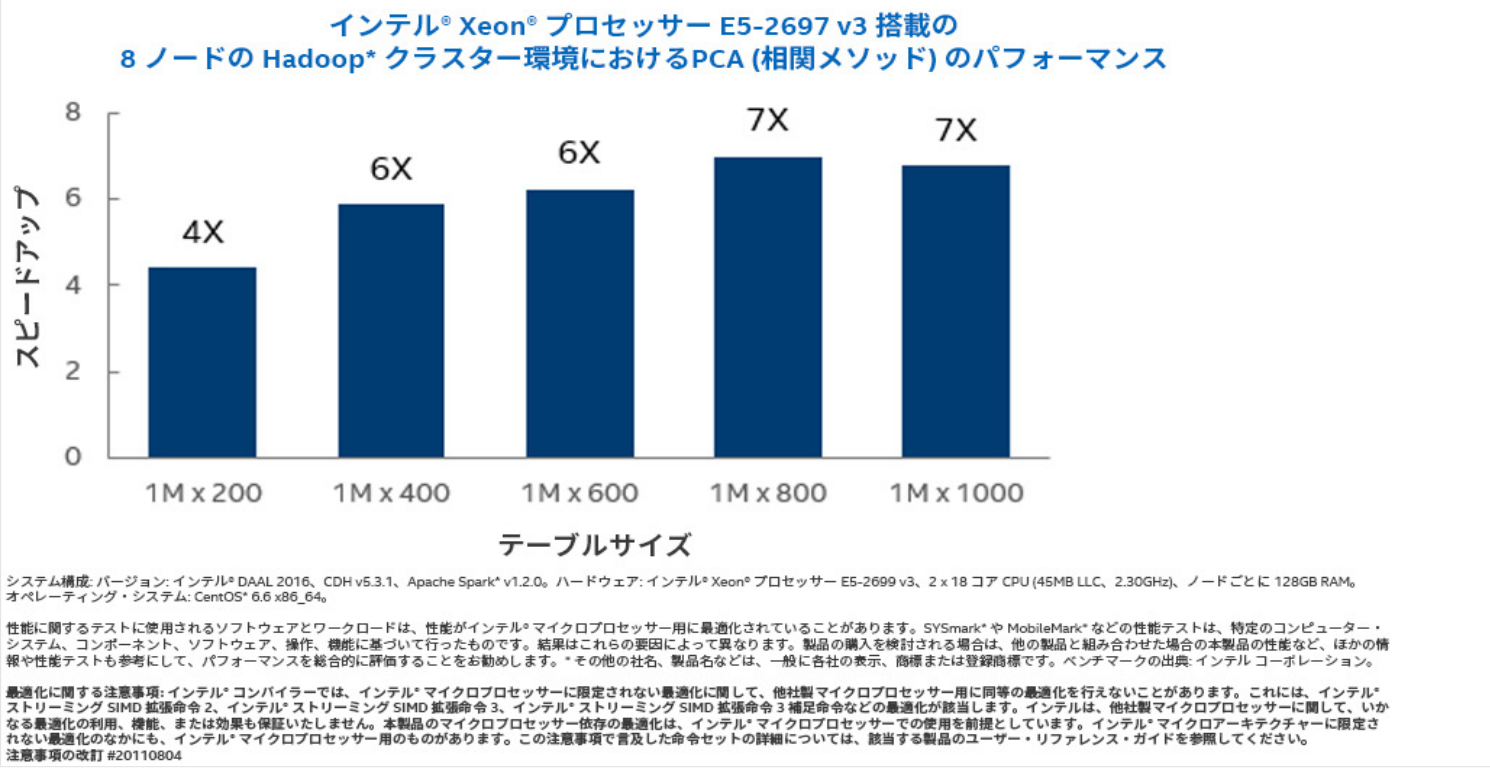
```
// ローカルノードで SVD メソッドを使用して PCA 分解を計算するアルゴリズムを作成
DistributedStep1Local pcaLocal = new
DistributedStep1Local(daalContext, Double.class, Method.svdDense);
// 入力データを設定
pcaLocal.input.set( InputId.data, ntData );
// ローカルノードで PCA を計算
PartialResult pres = pcaLocal.compute();
// マスターノードへ送るデータを準備
context.write(new IntWritable(0), new WriteableData(index, pres));
```

ステップ 2: マスターノード

```
// マスターノードで SVD メソッドを使用して PCA 分解を計算するアルゴリズムを作成
DistributedStep2Master pcaMaster = new DistributedStep2Master(daalContext,
Double.class, Method.svdDense);
// ローカルノードから受け取った入力データを追加
for (WriteableData value : values) {
      PartialResult pr = (PartialResult)value.getObject(daalContext);
      pcaMaster.input.add( MasterInputId.partialResults, pr);
}
// マスターノードで PCA を計算
pcaMaster.compute();
// 計算をファイナライズして PCA 結果を取得
Result res = pcaMaster.finalizeCompute();
HomogenNumericTable eigenValues  =
(HomogenNumericTable)res.get(ResultId.eigenValues );
```

パフォーマンス・データ

図 6 は、8 ノードの Hadoop* クラスター上でのインテル® DAAL と Spark* MLlib のパフォーマンス・ゲインを示したものです。

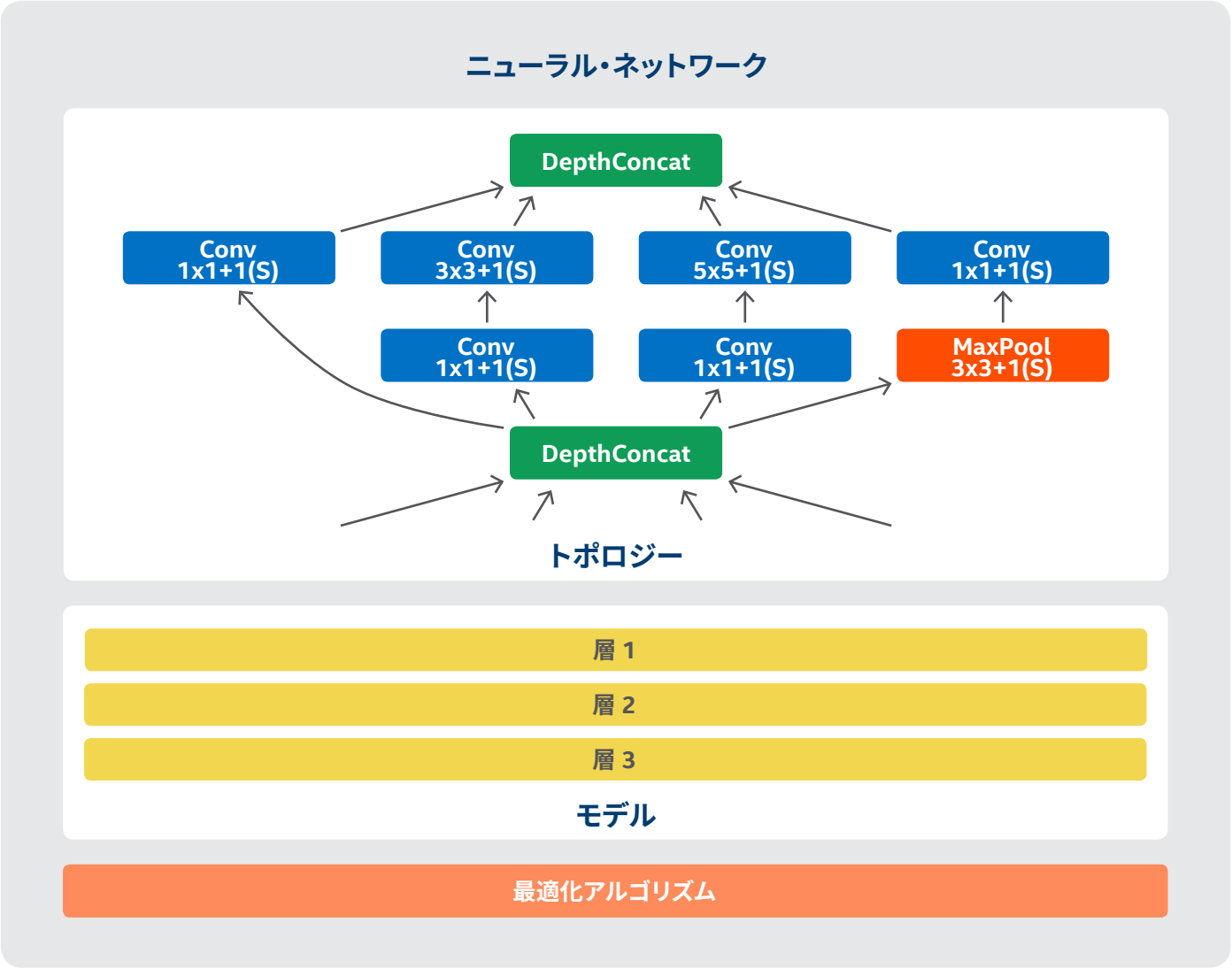


6 インテル® アーキテクチャー上でのインテル® DAAL と Spark* MLlib の PCA パフォーマンス

インテル® DAAL 2017 の新機能

インテル® DAAL 2017 では、多数の新機能が追加されています。特に、ニューラル・ネットワーク機能の実装では、画像分類やオブジェクト追跡から金融市場予測まで、さまざまなニューラル・ネットワーク・トポロジーをユーザーが簡単に作成し、実行できるコンポーネントが提供されています。多種多様なユースケースに対応できるように、ニューラル・ネットワーク向けのいくつかのビルディング・ブロックがインテル® DAAL に追加されました (**図 7**)。

- **層 :** NN ビルディング・ブロック。単一層のフォワードおよびバックワード・バージョン。
- **トポロジー :** NN 構成 (例 AlexNet) を表す構造。
- **モデル :** 定義されたトポロジー、パラメーター (重みとバイアス)、サービスルーチンに従って連結された層のセット。
- **最適化ソルバー :** 指定された目的関数に従って、フォワード - バックワード・パス後に重みとバイアスを更新。
- **NN:** NN 処理フローを管理するドライバー。トレーニング中、ドライバーはトポロジーに対してフォワード / バックワード・パスを実行し、最適化ソルバーを使用してパラメーターを更新します。スコアリング中、ドライバーはフォワードパスを適用して、予測結果を返します。
- **多次元データ構造 (テンソル):** 複雑なデータ (3D 画像のストリームなど) を表す構造。



7 インテル® DAAL のニューラル・ネットワーク向けのビルディング・ブロック

従来のマシンラーニング・アルゴリズムと同じ API を利用して、データ・サイエンティストは、インテル® アーキテクチャー上でパフォーマンスを向上するとともに、サポート・ベクトル・マシン (SVM) や線形回帰など、従来のマシンラーニング・アルゴリズムで作成されたモデルとニューラル・ネットワークで作成されたモデルの精度を比較できます。インテル® DAAL の機能により、ニューラル・ネットワークとその他のアルゴリズムを組み合わせること (例えば、弱学習器としてニューラル・ネットワーク・モデルをブースティング・アルゴリズムに挿入すること) が可能です。

ビッグデータとマシンラーニングの課題

ビッグデータ解析とマシンラーニングへの対応では、通常、問題を解決するために異なるプログラミング言語、ツール、ライブラリーが必要になります。データをさまざまなソースから取得したり、データがヘテロジニアスやスパースであったり、データにノイズが含まれていることはよくあることです。プロトタイプ生成時には Python* やその他のスクリプト言語がよく使用され、データフローの重要な部分の最適化には C/C++ 言語が使用されます。実際のデータは経時的に取得され、多くの場合 1 台のマシンのメモリーには収まりません。そのため、データを効率良く処理するには、MPI、Hadoop*、Spark* のような分散システムを使用する必要があります。

このことから、開発、テスト、サポートには、さまざまな分野の知識を備えた開発者が必要になります。多くのソリューションを 1 つのアプリケーションに統合するのには時間がかかります。結果、開発期間が大幅に延び、ビッグデータ解析の導入が遅れることになります。

インテル® DAAL は、これらすべての課題に対応し、ビッグデータを扱う上で開発者が直面する可能性のあるほとんどのユースケースをカバーするように設計されています。アルゴリズムだけでなくデータフロー全体を最適化するため、データソースからのデータの取得、前処理、変換、データマイニング、モデリング、検証、意思決定を含むすべてのデータ解析段階をカバーするビルディング・ブロックを提供します。同じ API を使用して、シングルノードから大規模クラスターにモデルをスケーリングすることが可能です。インテル® DAAL は、インテル® MKL とともに、ビッグデータ解析とマシンラーニングを有効利用できるように支援します。

インテル® DAAL の詳細 >