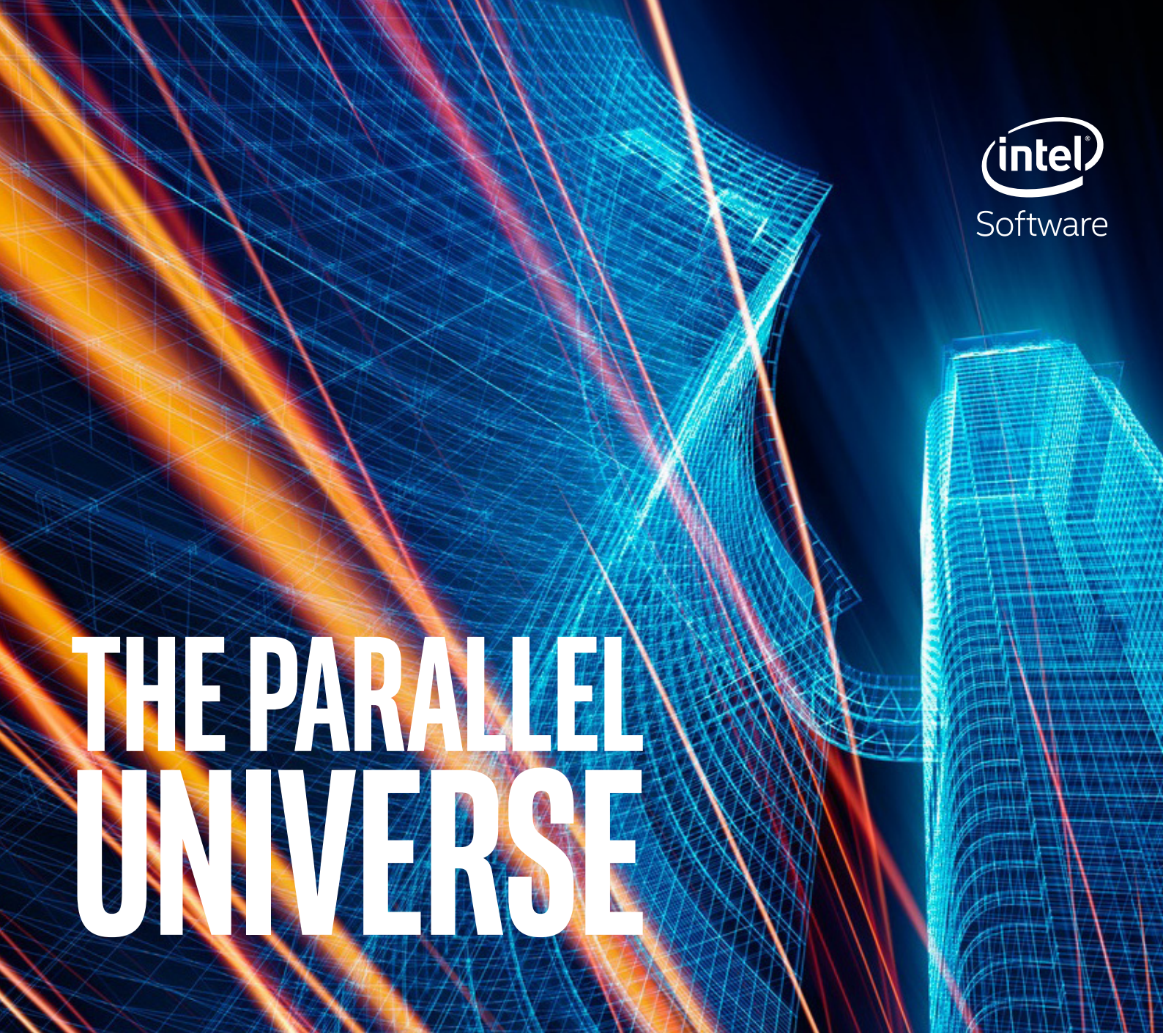


インテル® Xeon Phi™ プロセッサー向けのコードの現代化

インテル® VTune™ Amplifier XE による
Java* および Python* コードのプロファイル

データ解析およびマシンラーニング向け
パフォーマンス・ライブラリー

Issue
26
2016

目次

編集者からのメッセージ

Mike Lee インテル コーポレーション テクニカル、エンタープライズ、クラウド向けソフトウェア・ツール・マネージャー

マシンは私たちから何を学ぶのでしょうか？

プログラミングとは？皆さんご存知のとおり、コンピューター上でタスクを実行するため、開発者が記述する一連の明示的な論理命令です。しかし、これが変わりつつあります。近年、マシンラーニング分野は、科学、商業、産業、消費者によるコンピュート・プラットフォームと関連センサーの利用によって生成される膨大な量のデータから、価値のあるデータと詳細を抽出するため、重要性が増しつつあります。

マシンラーニング・アルゴリズムの実装には、特定のパターンを認識するようにトレーニングされたニューラル・ネットワークを使用します。例えば、プログラムに木を認識させる場合、枝、幹、葉を探すように明示的にプログラミングする代わりに、木のイメージを提供します。ニューラル・ネットワークが木のイメージを正しく認識できない場合は、プログラムを変更するのではなく、さらに多くの木のイメージを与えてトレーニングします。

開発者にとって今後の課題は、マシンラーニング手法の理解と実装を、最適なパフォーマンスと結果をプログラムに教える能力と組み合わせることです。

どのように取り掛かればよいのでしょうか？

最近リリースされたインテル® Parallel Studio XE 2017 を利用すると良いでしょう。インテル® Parallel Studio XE 2017 は、マシンラーニング関連の開発作業を支援する多くの新機能を提供します。新製品のインテル® Distribution for Python* は、インテル® マス・カーネル・ライブラリー (インテル® MKL)、インテル® Data Analytics Acceleration Library (インテル® DAAL)、インテル® MPI ライブラリー、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) などを利用して、NumPy*、SciPy*、scikit-learn のようなパッケージを高速化できます。インテル® MKL とインテル® DAAL は、最適化された畳み込み、プーリング、K 平均法、交互最小 2 乗 (ALS) を含むマシンラーニング開発向けの基本的なビルディング・ブロックを提供します。解析では、インテル® VTune™ Amplifier XE を利用することで、Python* コードと Python* と C/C++ 混在コードのプロファイルにより hotspot を特定できます。

本号の The Parallel Universe では、引き続きツールとライブラリーの概要とマシンラーニングでの利用法を紹介します。また、最近リリースされたインテル® Xeon Phi™ プロセッサー (開発コード名 Knights Landing) の最新機能を理解し、利用できるように支援します。従来のプログラミングを継続し、エクサスケール、さらにその先へ向かいつつある中、マシンラーニングの発展とそれらを支える手法は、プログラマーに新たな機会と課題をもたらします。次世代のコンピューターをトレーニングする頃には、プログラミングの概念が変わるかもしれません。

Mike Lee
2016 年 10 月

インテル® XEON PHI™ プロセッサー向けのコードの現代化

新しいインテル® Parallel Studio XE 2017 の機能によってもたらされる利点

Yolanda Chen 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
Udit Patidar インテル コーポレーション プロダクト・マーケティング・エンジニア

未来はここにあります。少なくとも、インテルの次世代のハードウェアについては、そう言えるでしょう。ハードウェア・アクセラレーターは長年、ハイパフォーマンス・コンピューティングにおける次なる目玉ともてはやされ、多くのベンダーが製品とサービスを提供してきました。しかし、アクセラレーターを最大限に活用するためのプログラミング・モデルは、習得が困難で、プログラマーは独自のコードに縛られがちです。

新しい**インテル® Xeon Phi™ プロセッサー** (開発コード名 : Knights Landing または KNL) は、この問題を軽減し、標準の CPU 機能を利用してアクセラレーターのパフォーマンスを引き出す、インテル初のプロセッサーです。インテル® Xeon Phi™ プロセッサーは、明示的なプログラムによる制御やコードのオフロードを必要としません。

そのため、OpenMP* を使用しているかどうかに関係なく、プログラムをシームレスに実行できます。**コードの現代化**と複数レベルでのスケーラブルな並列パフォーマンスの実現を支援するため、**インテル® Parallel Studio XE 2017** スイートには、強力な各種開発ツールが含まれています。

さらに、インテル® Xeon Phi™ プロセッサーはインテル® Xeon® プロセッサーとバイナリー互換であるため、現在 x86 アーキテクチャーで実行しているワークロードを、インテル® Xeon Phi™ プロセッサーの高度な並列処理向けにチューニングおよび最適化できます。

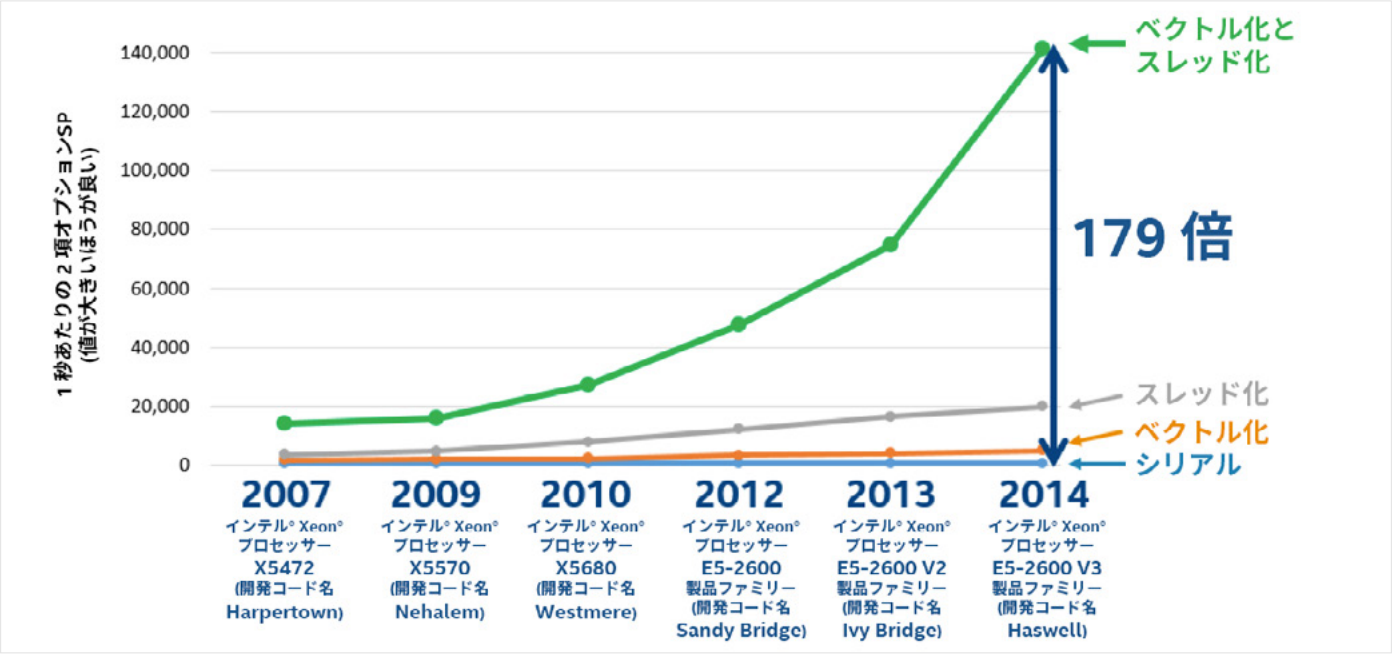
コードの現代化について

インテル® Xeon Phi™ プロセッサーは、8 つの倍精度浮動小数点数または 16 の単精度浮動小数点数を 512 ビット・ベクトルにパックできる、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) をサポートします。一般に、512 ビット・ベクトル拡張は、インテル ® AVX およびインテル® AVX2 命令よりも幅広い機能と高いパフォーマンスを提供します。ベクトルを効率良くパックし、無駄を最小限に抑えることは、言うのは簡単ですが、実行するのは困難です。ベクトル命令に加えて、ランタイム環境ではいくつかのスレッドを利用して、異なるレベルの並列処理が可能です。そのため、より複雑になります。コードの現代化の設計では、異なるレベルの並列処理を考慮する必要があります。

- **ベクトル並列処理 :** コア内で、異なるデータチャンクに対して同じ計算が実行されます。**SIMD** (Single Instruction, Multiple Data) **ベクトル化**と呼ばれます。SIMD ベクトル化を効率良く利用することで、コードのシリアル領域とスレッド並列領域の両方で利点が得られます。
- **スレッド並列処理 :** 複数のスレッドが共有メモリーを介してやり取りし、協調して与えられたタスクを実行します。

最新の OpenMP* 標準 (4.0 以降) では、明示的な **SIMD** ベクトル化とスレッド並列処理の両方を C/C++ および Fortran コードに追加可能な構造が用意されています。インテル® コンパイラーは、OpenMP* をサポートしています。

適切に利用することで、SIMD **ベクトル化**とスレッド化は、大幅なスピードアップをもたらします。**図 1** は、最新のインテル® Xeon® プロセッサー上で、シリアルコードと比較して 179 倍のスピードアップを示しています。インテル® Xeon Phi™ プロセッサーでも同様のパフォーマンスの向上が想定されます。



1 SIMD ベクトル化とスレッド並列処理を効率良く使用することで、2 項オプション・ワークロードでシーケンシャル・コードと比較してパフォーマンスが大幅に向上

インテル® Parallel Studio XE とコードの現代化プロセス

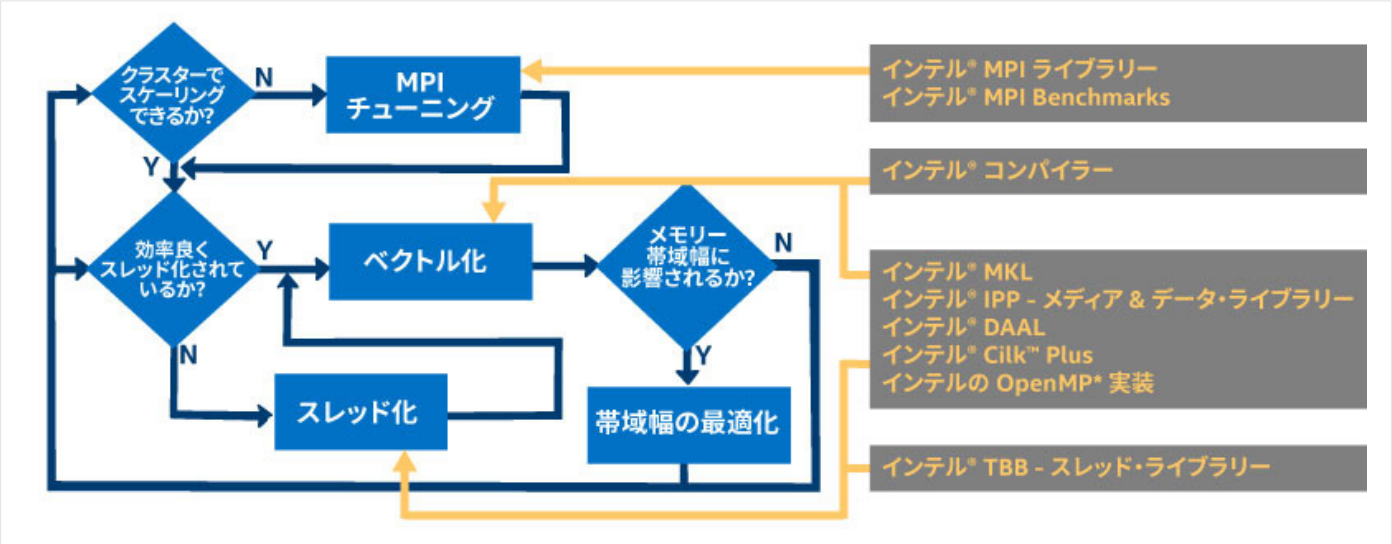
コードの現代化の最適化フレームワークについては、アプリケーション・パフォーマンスを向上する体系的なアプローチとして**こちらで**詳しく説明されています。このフレームワークは、インテル® Xeon Phi™ プロセッサーにも適用可能で、それぞれの最適化ステージで繰り返しアプリケーション・パフォーマンスを向上します。

1. **現在のコードとワークロードのプロファイル : インテル® VTune™ Amplifier XE** で hotspot を検出し、**インテル® Advisor** でベクトル化とスレッド化の可能性とヘッドルームを特定します。**インテル® コンパイラー**は最適なコードを生成し、**インテル® マス・カーネル・ライブラリー (インテル® MKL)**、**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)**、**インテル® インテグレーテッド・パフォーマンス・プリミティブ (インテル® IPP)** などの最適化されたライブラリーを適用します。

2. **スカラーコードの最適化 :** 正しい精度のデータ型で、適切な関数を使用し、コンパイル時に精度に関するコンパイラー・オプションを設定していることを確認します。

3. **明示的な SIMD ベクトル化 :** OpenMP* ベースの SIMD ベクトル化機能とデータレイアウトの最適化を併用します。適切なデータ構造を定義し、SIMD Data Layout Template (**Parallel Universe 24 号**を参照) を使用して **C++ コードを構造体配列 (AOS) から配列構造体 (SOA) に変換**します。

4. **スレッド並列処理 :** OpenMP* と環境変数を利用して、コアに対し適切なスレッド・アフィニティーを設定します。**インテル® Inspector** でデバッグし、スケーリングの問題を引き起こすスレッドエラーがないことを確認します。スケーリングの問題は通常、スレッドの同期や非効率なメモリー使用により生じます。



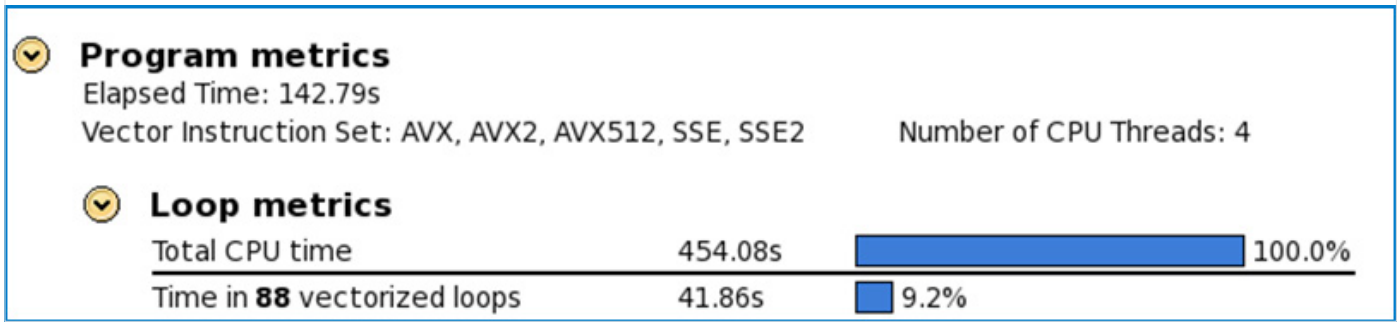
2 インテル® Xeon Phi™ プロセッサー向けのコードの現代化

インテル® Parallel Studio XE の各種コンポーネント (コンパイラー、最適化ライブラリー、パフォーマンス・アナライザー / プロファイラー) は、インテル® Xeon Phi™ プロセッサーの可能性を最大限に引き出せるように、コードの現代化を支援します。**図 2** は、インテル® Xeon Phi™ プロセッサー向けのコードの現代化プロセスと、各プロセスを支援するインテルの各種ツールを示します。クラスターでない場合は、最初の 2 つのボックスをスキップして、「効率良くスレッド化されているか ?」から開始します。

インテル® コンパイラーは、インテル® Xeon® プロセッサーとインテル® Xeon Phi™ プロセッサー向けの最新のインテル® AVX-512 命令をサポートします。また、最新の C++ および Fortran 標準規格のサポートと下位互換性も提供します。さらに、OpenMP* 4.0 および 4.5 を利用して、明示的な SIMD ベクトル化とスレッド化により、大幅なパフォーマンスの向上を達成できます (**図 1**)。(インテル® Xeon Phi™ プロセッサー向けのコンパイラー・サポートについては後述します。)

インテル® VTune™ Amplifier XE 2017 は、インテル® Xeon Phi™ プロセッサー向けのいくつかの重要な最適化を提示します。インテル® Xeon Phi™ プロセッサーでは、主要機能である MCDRAM の適切な使用法を決定する必要があります。インテル® VTune™ Amplifier XE のパイプライン / キャッシュ、メモリー解析、スケーラビリティー解析は、次の作業を支援します。

- MCDRAM に配置するデータ構造の決定
- メモリー階層でパフォーマンスの問題を表示
- DRAM と MCDRAM の帯域幅を測定



3 インテル® Xeon Phi™ プロセッサーで実際のワークロードを実行したインテル® Advisor の Summary 出力例

さらに、シリアル時間と並列時間を測定し、インバランスとオーバーヘッド・コストを特定することで、OpenMP* のスケーラビリティーの問題にも対応できます。最後に、インテル® VTune™ Amplifier XE は、コア・パイプライン上のコードの効率を示すことで、インテル® Xeon Phi™ プロセッサー・マイクロアーキテクチャーの利用効率を明らかにします。

インテル® Advisor は、コードの現代化において重要な役割を果たします。インテル® Advisor を使用することで、ヘッドルームを簡単に特定し、インテル® Xeon Phi™ プロセッサー固有のインテル® AVX-512 命令向けに最適化できます。例えば、インテル® Xeon Phi™ プロセッサーで実際のワークロードを実行したインテル®Advisor の Summary (図 3) は、多くのカーネルがベクトル化されているにもかかわらず、まだ多くの改善の余地があることを示しています。

インテル® Xeon® プロセッサーとインテル® Xeon Phi™ プロセッサーに高速な数学機能を提供する**インテル® MKL** は、コードの現代化の主要コンポーネントであり、シングルコア (明示的なベクトル化) からマルチコア (スレッド化) まで幅広くパフォーマンスの自動スケーリングを達成するのに不可欠なツールです。さらに、新しい**インテル® Distribution for Python*** は、内部でインテル ® MKL を使用して最適化されています。ホスト - オフロードモデルで実行する場合、インテル® MKL はホスト CPU とインテル® Xeon Phi™ コプロセッサー間の最適なロードバランスを特定します。(インテル® Distribution for Python* の詳細は**こちら**を参照してください。)

誕生から 10 年を迎えた**インテル® TBB** は、ハイパフォーマンスでスケーラブルな並列アプリケーションを作成するための、広く使用されている実績ある C++ ライブラリーです。(インテル® TBB による多数の最適化については、**Parallel Universe 特別号** (英語) を参照してください。)

インテル® IPP は、すぐに使用可能な、プロセッサー向けに最適化されたビルディング・ブロックで、画像、信号、データ、および暗号化計算処理を高速化します。ルーチン群は、インテル® AVX-512 を含む最新の命令セット向けに最適化されているため、コンパイラーによるインテル® Xeon Phi™ プロセッサー向けの最適化を単独で使用する場合よりもパフォーマンスを向上できます。

インテル® Xeon Phi™ プロセッサー向けのインテル® コンパイラー 17.0 の拡張機能

ここでは、インテル® コンパイラーによってもたらされるベクトル化の可能性について詳しく説明します。インテル® Xeon Phi™ プロセッサーに関して、標準の C/C++ および Fortran アプリケーションには、インテル® コンパイラーの一般的な機能が適用されます。中でも、インテル® Xeon Phi™ プロセッサーで追加されたインテル® AVX-512 命令セットに関する拡張が重要です。

インテル® コンパイラー 17.0 では、次の点においてベクトル化のサポートが拡張されています。

- 仮想関数を含むベクトル関数の間接呼び出し (**Parallel Universe 25 号**を参照)
- OpenMP* 4.5 の配列と部分配列のリダクション・サポート
- 間接メモリー参照のプリフェッチ

また、-xMIC-AVX512 コンパイラー・オプションにより、インテル® Xeon Phi™ プロセッサー上で既存のインテル® AVX-512 がサポートされます。

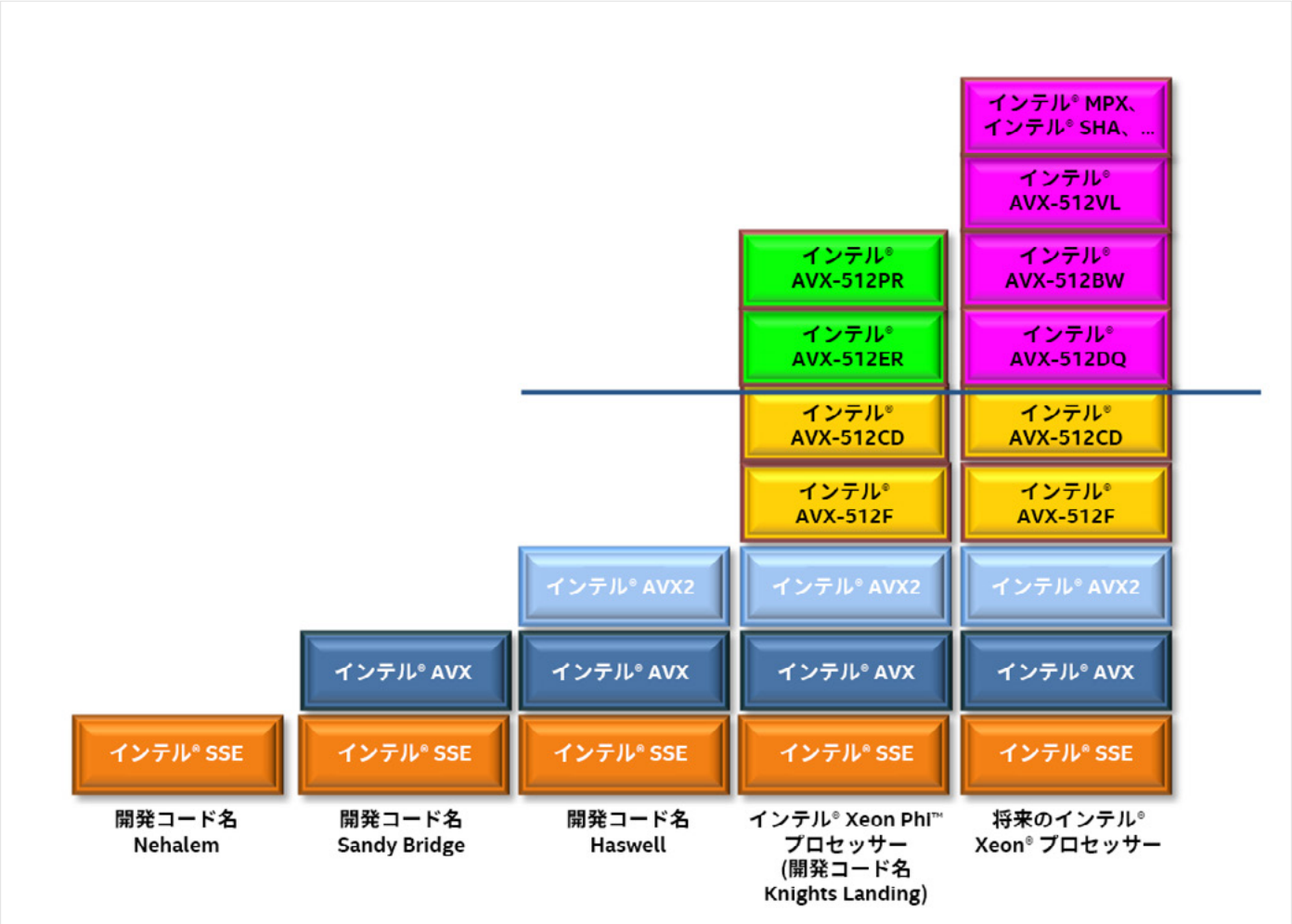
インテル® AVX-512 向けのベクトル化

512 ビット SIMD 拡張命令のベースは、インテル® AVX-512 基本命令と呼ばれます。インテル® AVX および インテル® AVX2 ファミリーの SIMD 命令を拡張したもので、512 ビット・ベクトル・レジスター、64 ビット・モードの最大 32 ベクトルレジスター、マスクレジスターを利用した条件処理をサポートする新しいエンコーディング・スキームでエンコードされています。

インテル® AVX-512 ファミリーは、特定のアプリケーション・ドメイン向けに、いくつかの追加の 512 ビット拡張命令を提供します。

- **インテル® AVX-512 指数および逆数命令** (インテル® AVX-512ER): 特定の超越関数向け
- **インテル® AVX-512 プリフェッチ命令** (インテル® AVX-512PR): 特定のプリフェッチ操作
- **今後拡張される命令** : SHA、MPX 向けなど

図 4 に示すように、これらの命令のいくつかは、インテル® Xeon Phi™ プロセッサー (開発コード名 Knights Landing 以降) でサポートされており、今後拡張される命令は将来のインテル® Xeon® プロセッサーでサポートされる予定です。



4 インテル® Xeon Phi™ プロセッサーと将来のインテル® Xeon® プロセッサーは多くの命令セットを共有

1. インテル® AVX-512 基本命令

インテル® AVX-512 基本命令は、インテル® AVX およびインテル® AVX2 の拡張です。これらの命令が 512 ビット・ベクトルで動作し、EVEX プリフィクスでエンコードされた命令セット拡張が 512 ビット未満のベクトル長で動作するようにサポートします。そのため、インテル® Xeon® プロセッサー向けにベクトル化されたループは、インテル® Xeon Phi™ プロセッサーでもベクトル化されます。ベクトル長は、ループカウントと配列の長さに応じて拡張されます。

インテル® Xeon® プロセッサー向けの既存のアプリケーションをインテル® AVX-512 対応のインテル® Xeon Phi™ プロセッサーで実行するには、**-xMIC-AVX512** 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、生成されたバイナリーをインテル® Xeon Phi™ プロセッサー・ベースのシステムにコピーするだけです。

2. インテル® AVX-512PR

インテル® AVX-512PR とインテル® AVX-512ER は、インテル® Xeon Phi ™ プロセッサーでのみ利用可能な命令セットです。インテル® AVX-512PR は、ギャザー / スキャッターと PREFETCHWT1 向けの新しいプリフェッチ命令セットです。インテル® コンパイラー 17.0 でこの命令を有効にするには、次のコンパイラー・オプションを指定します。

-O3 -xmic-avx512 -qopt-prefetch=<n>

- **n=0**: **-qopt-prefetch** オプションを省略した場合のデフォルト値 (プリフェッチは発行されません)。
- **n=2**: "**n**" 引数を指定せずに **-qopt-prefetch** を指定した場合のデフォルト値。ハードウェア・プリフェッチが適切でないとコンパイラーが判断した直接参照にのみプリフェッチが挿入されます。
- **n=3**: ハードウェア・プリフェッチに関係なく、すべての直接メモリー参照でプリフェッチが有効になります。
- **n=5**: すべての直接および間接プリフェッチでプリフェッチが有効になります。間接プリフェッチには、インテル® AVX512-PF の gatherpf 命令を使用します。

#pragma prefetch var:hint:distance プラグマを使用して、コンパイラーが特定のメモリー参照に対してプリフェッチを発行するように強制することもできます。これにより、**-qopt-prefetch=2** または **3** を設定すると、プラグマにより指定された直接または間接参照イベントに対してプリフェッチが発行されます。

例えば、次の C++ ループを **-O3 -xmic-avx512 -qopt-prefetch=5 -qopt-report5** オプションを指定してコンパイルします。

```
//pragma_prefetch var:hint:distance
#pragma prefetch A:1:3
#pragma vector aligned
#pragma simd
    for(int i=0; i<n; i++)  {
    C[i] = A[B[i]];
      }
```

間接メモリー参照に対して、ギャザー / スキャッター・プリフェッチの生成に関する、**図 5** のようなリマークが出力されます。

```
   remark #25018: Total number of lines prefetched=16
   remark #25019: Number of spatial prefetches=15, dist=8
   remark #25033: Number of indirect prefetches=1, dist=2
   remark #25150: Using directive-based hint=1, distance=3 for indirect memory
reference     [ testpr.cpp(8,19) ]
   remark #25540: Using gather/scatter prefetch for indirect memory reference,
dist=3    [ testpr.cpp(8,19) ]
   remark #25143: Inserting bound-check around lfetches for loop
LOOP END
```

5 コンパイラーによる最適化レポートに出力されたギャザー/スキャッター・プリフェッチに関するリマーク

3. インテル® AVX-512ER

インテル® AVX-512ER は、指数、逆数、逆平方根関数の高速で高精度な近似命令を提供します。インテル® AVX-512 の指数命令 (VEXP*) は、パックド倍精度または単精度浮動小数点値の指数 2^x を相対誤差 2^-23 未満で計算します。これらの命令は、インテル® Xeon Phi™ プロセッサーでのみ利用できます。

現在、これらの命令は 512 ビットの ZMM レジスターのみサポートします。つまり、すべてのベクトル幅を使用して **exp** を計算するには、8 つの倍精度または 16 の単精度浮動小数点数が必要です。そうでない場合、**exp** 関数の SVML (Short Vector Mathematical Library) バージョンが呼び出されます。例えば、**図 6** の Fortran ループは、**arrayB(i)** が単精度の場合、VEXP 命令を生成しません。

```
do i =1,8
      arrayC(i) = arrayA(i)*exp(arrayB(i))
   end do
```

6 Fortran ループ

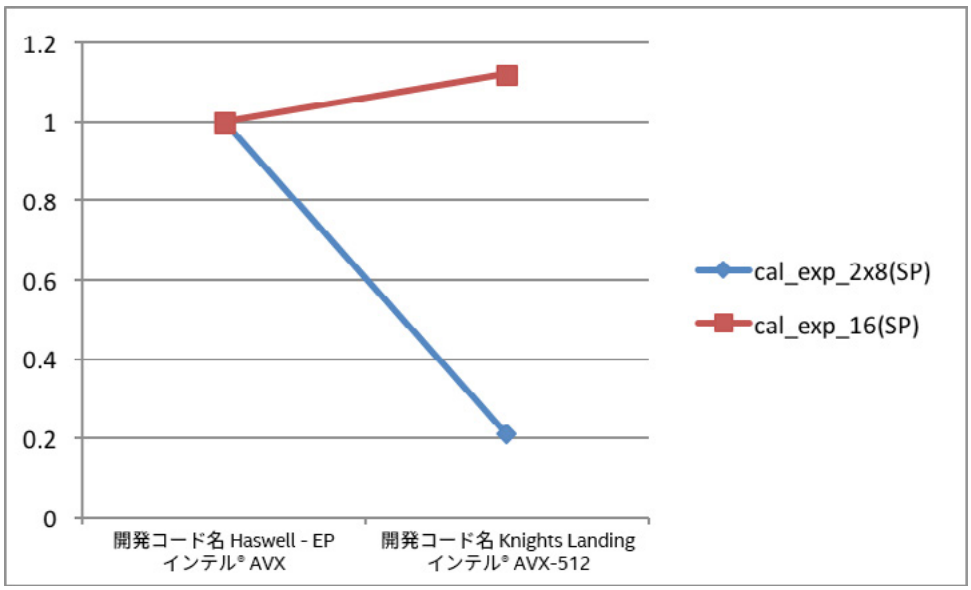
この制限に対処するため、同様の **exp** 呼び出しを持つ小さな配列をマージして、より大きな配列にします。**図 7** では、ループカウントを ZMM ベクトルに拡張できます。VEXP 命令のハードウェア実装は、インテル® Xeon Phi™ プロセッサー上では非常に効率が良いため、ベクトル化により利点が得られます。**図 7** は、Fortran の例です。

```
Module testER
implicit none
real, parameter, dimension(8) :: arrayA1 = (/2.0,1.5,1.37,2.4,3.3,4.9,5.1,0.0/)
real, parameter, dimension(8) :: arrayA2 = (/0.3,7.1,4.1,3.8,9.1,0.5,0.0,1.2/)
real, parameter, dimension(8) :: arrayB1 = (/8.0,1.2,1.4,1.7,2.58,3.4,5.0,7.1/)
real, parameter, dimension(8) :: arrayB2 = (/0.6,1.3,2.8,9.6,2.3,1.5,0.2,0.3/)
real, parameter, dimension(16) :: arrayA = &
(/2.0,1.5,1.37,2.4,3.3,4.9,5.1,0.0,0.3,7.1,4.1,3.8,9.1,0.5,0.0,1.2/)
real, parameter, dimension(16) :: arrayB = &
(/8.0,1.2,1.4,1.7,2.8,3.4,5.0,7.1,0.6,1.3,2.8,9.6,2.3,1.5,0.2,0.3/)
!DIR$ ATTRIBUTES ALIGN : 64 :: arrayA, arrayB, arrayA2, arrayB2

contains
subroutine cal_exp_16(arrayC)
real, dimension(16) :: arrayC
integer i

!dec$ vector aligned
do i=1,16
  arrayC(i) = exp(arrayA(i))*arrayB(i)
enddo
end subroutine

subroutine cal_exp_2x8(arrayC)
real, dimension(16) :: arrayC
integer i

!dec$ vector aligned
do i=1,8
  arrayC(i) = exp(arrayA1(i))*arrayB1(i)
end do

do i=1,8
  arrayC(i+8) = exp(arrayA2(i))*arrayB2(i)
enddo

end subroutine
end module
```

7 Fortran の例

配列 **arrayA** は、**arrayA1** と **arrayA2** をマージしたものです。ルーチン **cal_exp_16** は、**arrayA** にある 16 の単精度数の **exp** を計算し、結果と別の 16 の単精度数の乗算を行います。ルーチン **cal_exp_2x8** は、**arrayA1** と **arrayA2** を 2 つの小さなループで別々に計算します。テストの結果、**図 8** に示すように、**cal_exp_16** はインテル® Xeon Phi™ プロセッサー上で **cal_exp_2x8** よりも 5 倍以上高速に動作しました。使用したコンパイラー・オプションは、開発コード名 Haswell-EP では "**-xCORE-AVX2 -O2**"、インテル® Xeon Phi™ プロセッサーでは "**-xMIC-AVX512 -O2**" です。これはシングルコアのパフォーマンスで、開発コード名 Haswell-EP のクロック周波数は開発コード名 Knights Landing の 1.6 倍です。VEXP が有効な **cal_exp_16** バイナリーは、開発コード名 Haswell-EP (HSW-EP) のインテル® AVX2 よりも 1.12 倍高速です。

この例から、インテル® Xeon Phi™ プロセッサー上での 256 ビット・ベクトルを使用した **exp** の SVML バージョンの呼び出しは、インテル® Xeon® プロセッサーよりも効率が悪いことが分かります。インテル® Xeon Phi™ プロセッサー上で 512 ビット・ベクトルを使用するため小さな配列をマージすることで、コンパイラーが高速なインテル® AVX-512ER 命令を生成することができ、パフォーマンスが大幅に向上します。



開発コード名 Haswell-EP (HSW-EP)　インテル® Xeon® プロセッサー E5-2699 v3 @ 2.30GHz
開発コード名 Knights Landing (KNL)　インテル® Xeon Phi™ プロセッサー 7250 @ 1.40GHz

8 ユニットテストのパフォーマンス (スピードアップ)

4. インテル® AVX-512CD

インテル® AVX-512CD は、インテル® Xeon Phi™ プロセッサーで追加された新しい命令で、将来のインテル® Xeon® プロセッサーでもサポートされる予定です。ベクトル内の競合を検出できるため、ループ内の特定のケースの依存性を解決するのに役立ちます。一般的なスキームは、" ヒストグラム・アップデート " と特徴付けることができます。メモリー位置を読み取り、操作を行い、メモリー位置へ格納します。**図 9** は、このアクセスパターンを含むサンプル C コードです。

```
for (i=0; i < 512; i++)
    histo[key[i]] += 1;
```

9 アクセスパターン

key[n] と **key[m]** が同じ場合、配列 **histo** は正しい順序で読み取り / 書き込みを行う必要があります。コンパイラーの自動ベクトル化は、潜在的な依存性が存在する場合であっても、競合検出命令を生成し、このようなループをベクトル化します。

このアクセスパターンは、複数のヒストグラム・アップデートを含むより複雑なケースにも拡張できます (**図 10**)。

```
for (int j = 0; j < 512; j++)
       const int j1 = (int)ind1[j];
       const int j2 = (int)ind2[j];

for (int i = 0; i < 512; ++) {
        const int k1 = (int)key1[i];
        const int k2 = (int)key2 [i];

        histo[j1][k1] += a[i] ;
        histo[j2][k1] += b[i];
        histo[j1][k2] += c[i] ;
        histo[j1][k2] += d[i] ;
 }
```

10 複雑なアクセスパターン

4 つのヒストグラム・アップデートと 2D インデックスを含むこのループは、インテル® AVX-512CD を利用することで、コンパイラーによる自動ベクトル化が可能です。

最後に

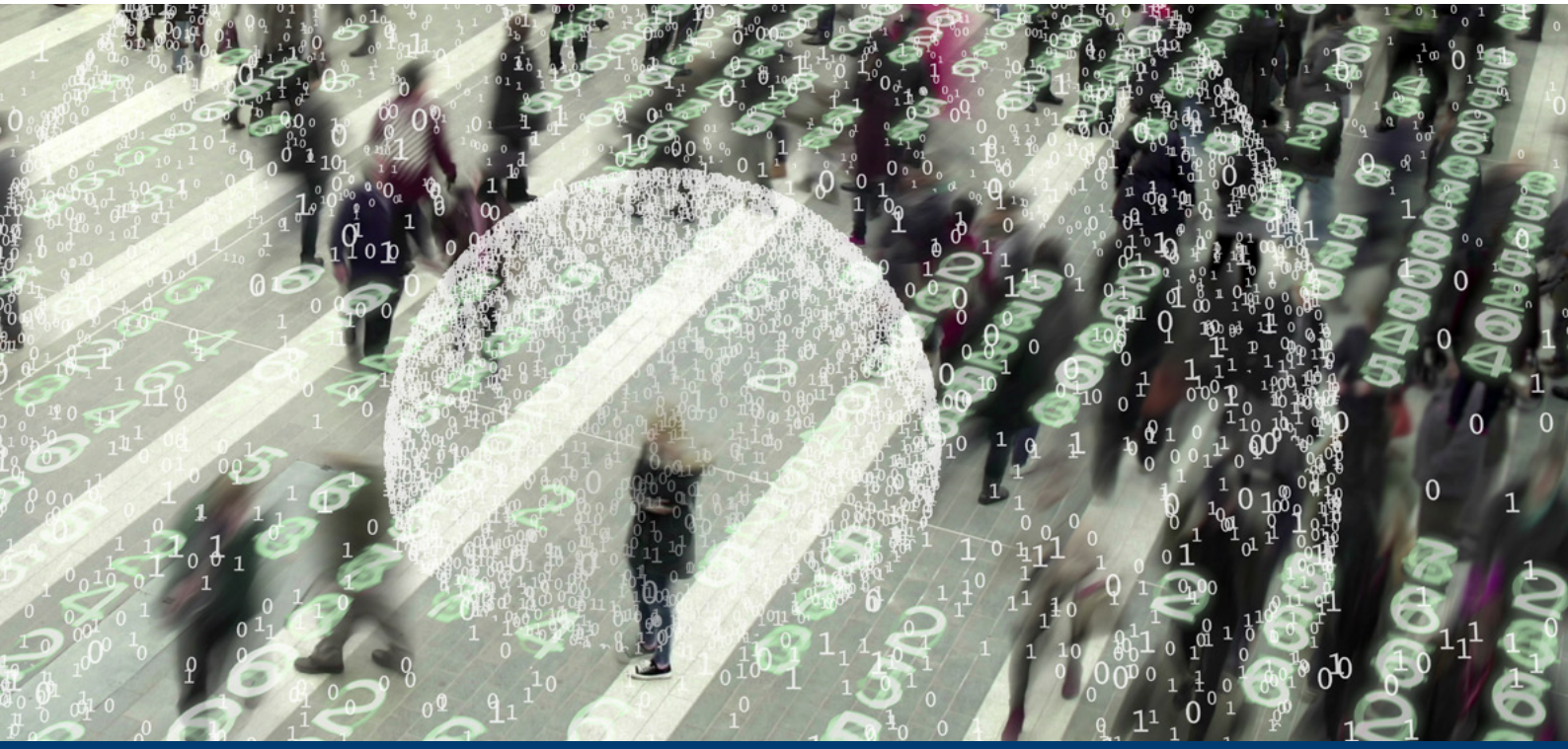
この記事では、ツールの概要とコンパイラーの詳細とともに、インテル® Parallel Studio 2017 のいくつかの斬新的な (そして、革新的とも言える) 新機能を紹介しました。これらの機能は、より深い洞察を提供するインテルの最新のプロセッサーであるインテル® Xeon Phi™ プロセッサーの性能を引き出すため、既存および新しい C/C++/Fortran コードをベクトル化し、マルチスレッド化するのに役立ちます。コードの現代化に今すぐ取り掛かりましょう。

関連情報

インテル® アーキテクチャー命令セット拡張プログラミング・リファレンス (英語)

コードの現代化 (英語)

インテル® Xeon Phi™ プロセッサーのパフォーマンスの証明 (英語)



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。

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

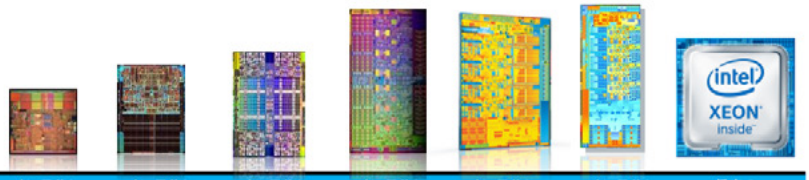
この 10 年間、マシンラーニングは急激に成長しました。この成長は、膨大な量のデータを生成するインターネットによって後押しされました。データからパターンを抽出し、その情報を適用して予測するため、新しいアプローチとアルゴリズムが開発されました。そして、計算能力の指数関数的成長により、それらのアルゴリズムを膨大なデータセットに適用して、有用な予測を行うことが可能になりました。

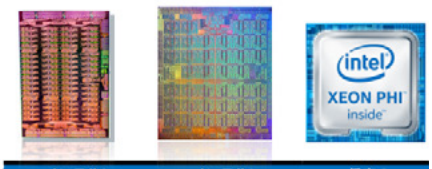
ディープ・ニューラル・ネットワーク (DNN) は、マシンラーニング分野の最先端のアルゴリズムです。1990 年代後半に業界での採用が進み、当初は銀行小切手の手書き文字認識などのタスクに利用されていました。ディープ・ニューラル・ネットワークは、このタスクにおいて人と同等またはそれ以上の能力を提供します。今日 DNN は、画像認識、ビデオおよび自然言語処理、自動運転などに必要な複雑な視覚理解の問題の解決に使用されています。DNN には、非常に多くの計算リソースと処理データが必要です。一例として、最新の画像認識トポロジー AlexNet のトレーニングには最新の計算システムで数日を要し、1400 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。

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?!

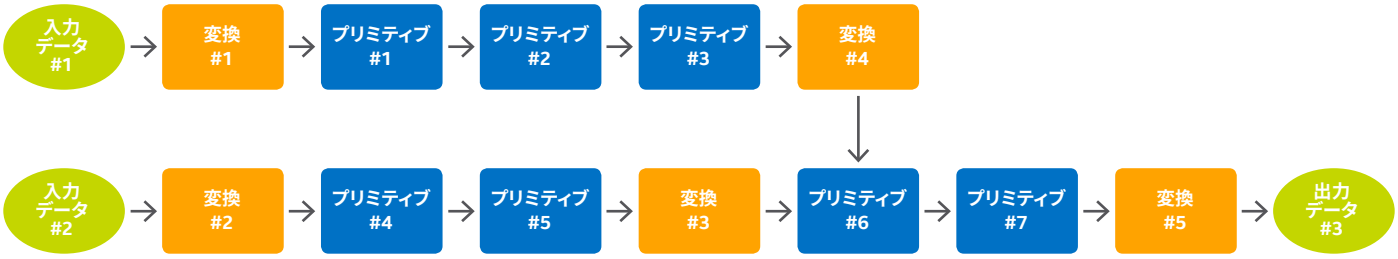
LENNY TRAN >

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。



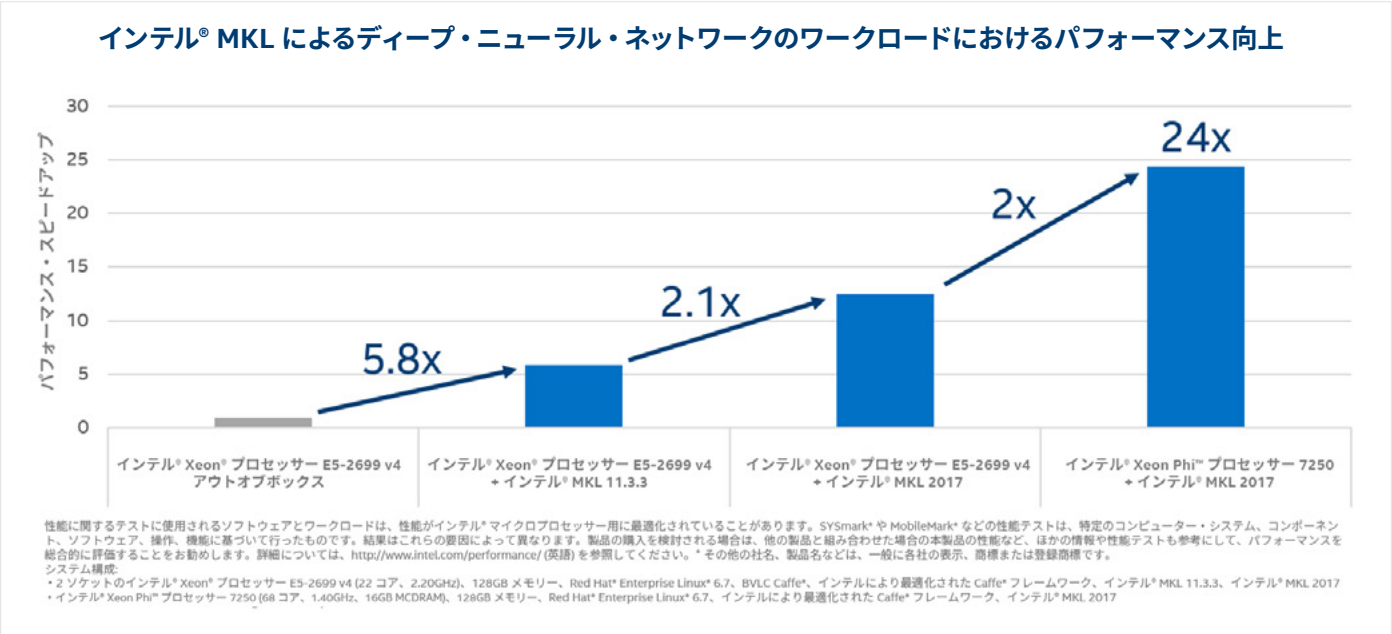
2 入力ステップ

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。

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。

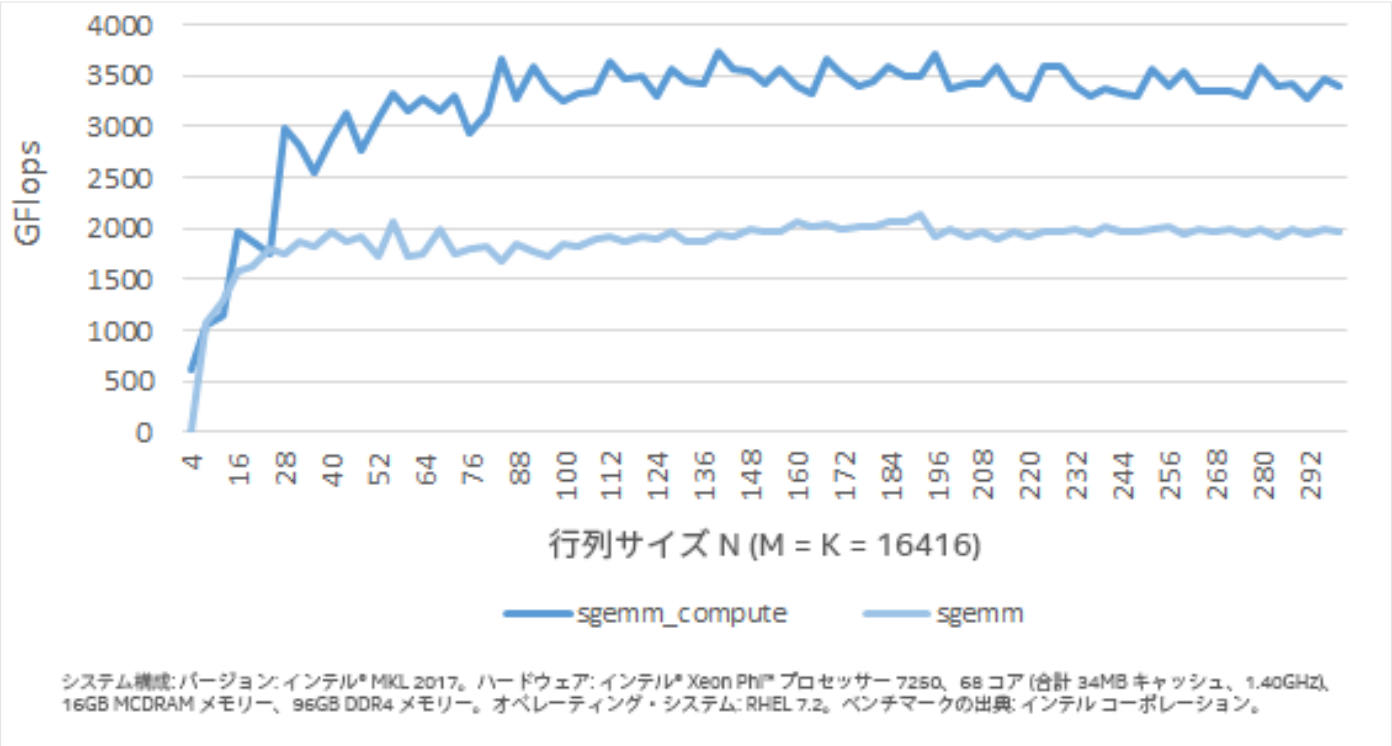
この記事の続きはこちら (英語) でご覧になれます。>

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。

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)

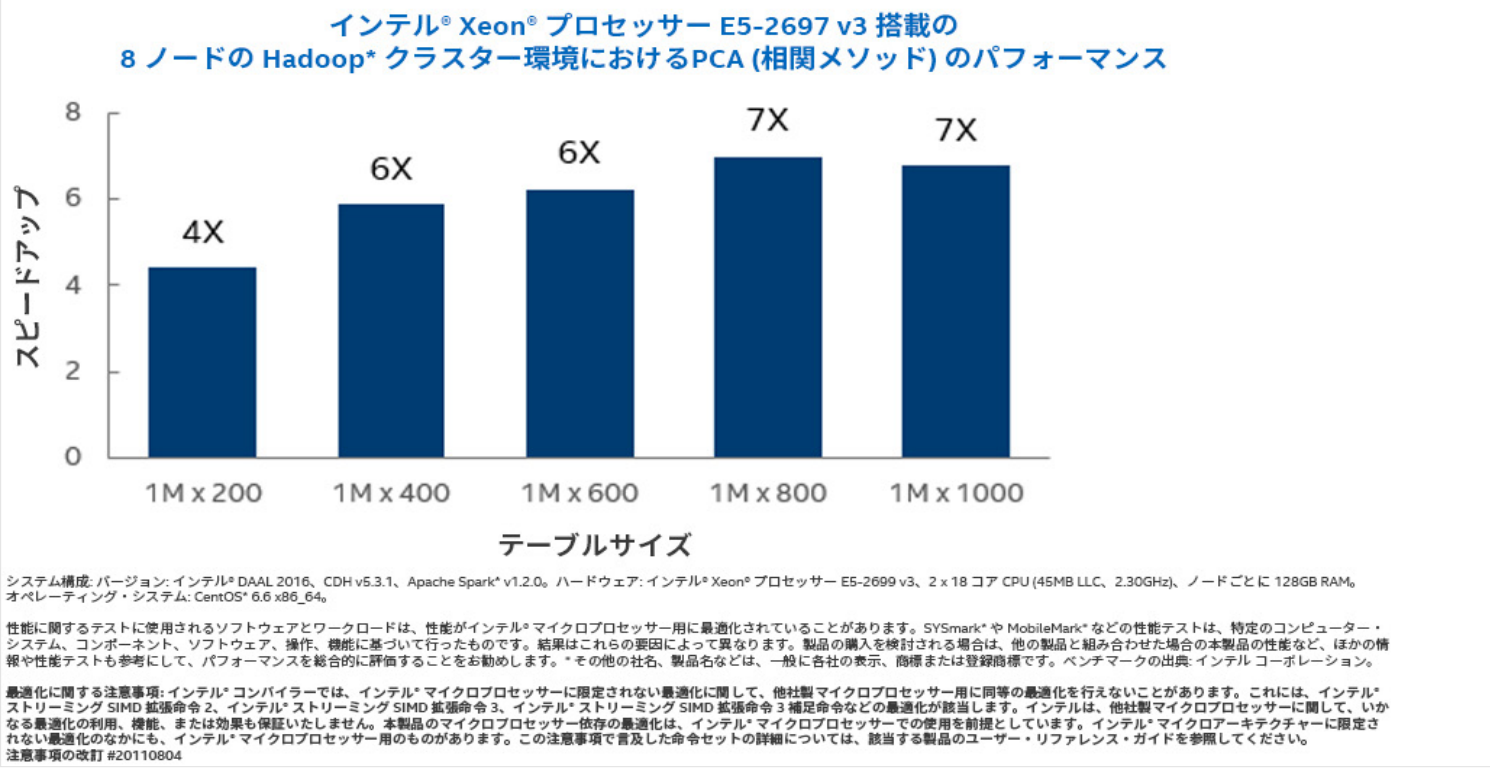
```
// ローカルノードで SVD メソッドを使用して PCA 分解を計算するアルゴリズムを作成
DistributedStep1Local pcaLocal = new
DistributedStep1Local(daalContext, Double.class, Method.svdDense);
// 入力データを設定
pcaLocal.input.set( InputId.data, ntData );
// ローカルノードで PCA を計算
PartialResult pres = pcaLocal.compute();
// マスターノードへ送るデータを準備
context.write(new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

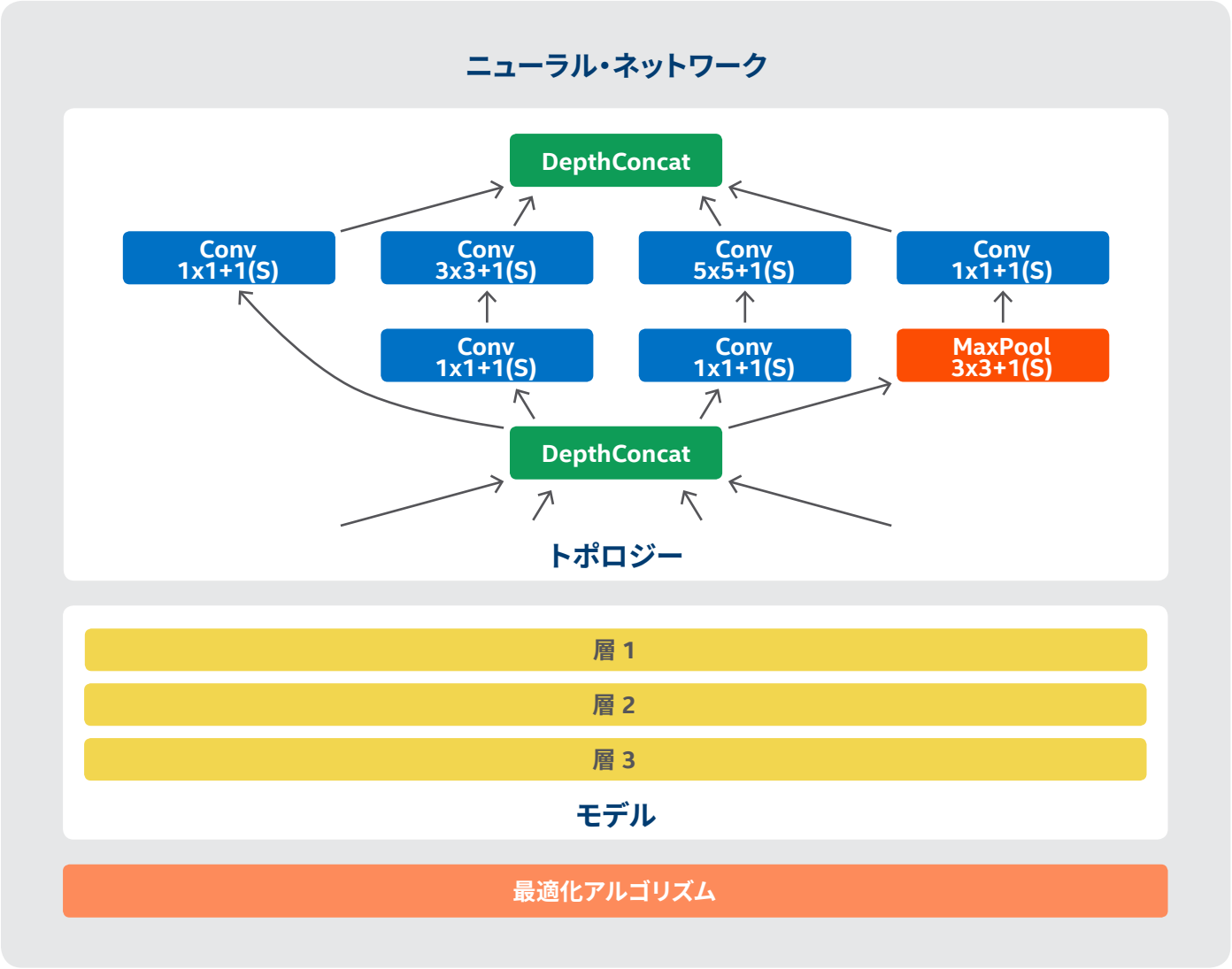


6 インテル® アーキテクチャー上でのインテル® DAAL と Spark* MLlib の PCA パフォーマンス

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

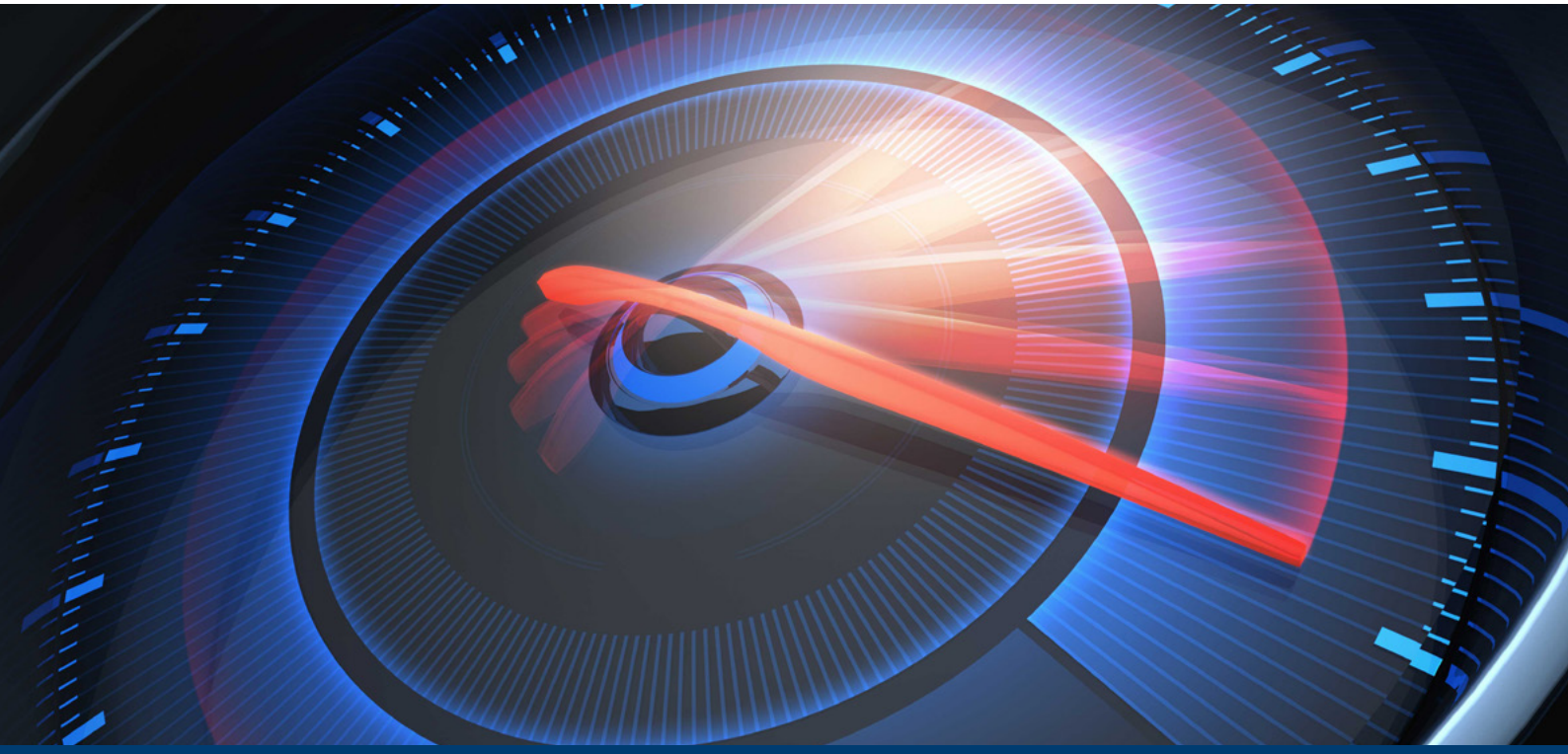
従来のマシンラーニング・アルゴリズムと同じ API を利用して、データ・サイエンティストは、インテル® アーキテクチャー上でパフォーマンスを向上するとともに、サポート・ベクトル・マシン (SVM) や線形回帰など、従来のマシンラーニング・アルゴリズムで作成されたモデルとニューラル・ネットワークで作成されたモデルの精度を比較できます。インテル® DAAL の機能により、ニューラル・ネットワークとその他のアルゴリズムを組み合わせること (例えば、弱学習器としてニューラル・ネットワーク・モデルをブースティング・アルゴリズムに挿入すること) が可能です。

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。



マシンラーニングにおける PYTHON* パフォーマンスの壁を乗り越える

Python* マシンラーニング・アプリケーションの高速化と最適化

Vasily Litvinov、Viktoriya Fedotova、Anton Malakhov、Aleksei Fedotov、Ruslan Israfilov、Christopher Hogan インテル コーポレーション ソフトウェア・エンジニア

協調フィルタリングによるパフォーマンスの高速化

インテルは、革新的なツール、手法、最適化により、Python* パフォーマンスを高速化するため、マルチコアおよび **SIMD** 並列処理向けのハイパフォーマンス・ライブラリー、プロファイラー、拡張サポートを提供します。ここでは、協調フィルタリングを使用して、実際のマシンラーニング・アプリケーションのパフォーマンスを高速化します。

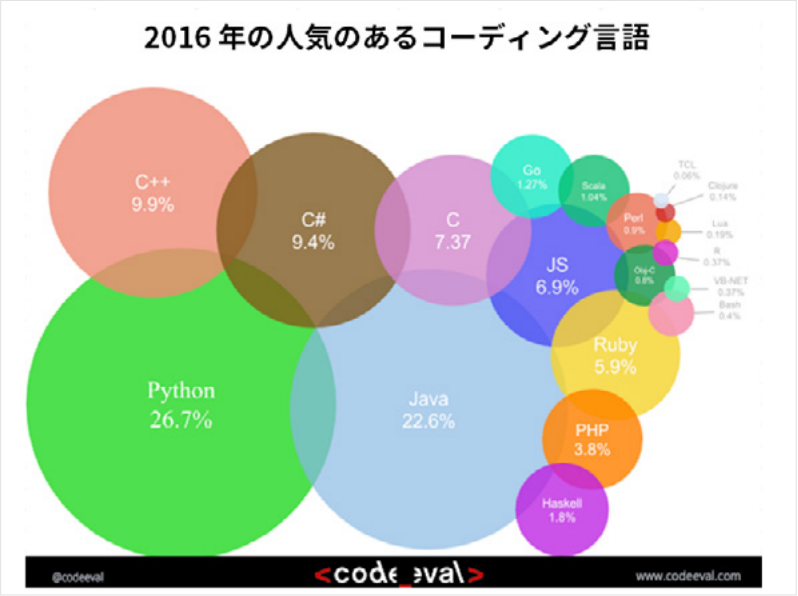
結果は、これらの最適化を利用することで Python* でネイティブコードに近いパフォーマンスを達成できることを示しています。つまり、パフォーマンスを向上するため、C/C++ でコードを書き直す必要はありません。

問題を理解する

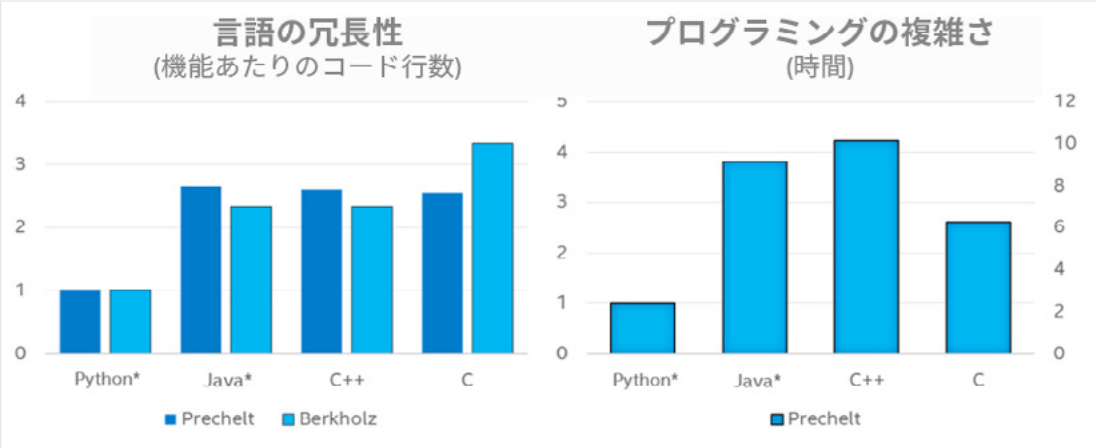
Python* の人気

Python* は、その生産性の高さから、着実に採用が進んでいます。2016 年の **CodeEval.com** の投票では、Python* が開発者に最も人気のある言語に選ばれました。2 位は Java*、3 位は C++ でした (**図 1**)。[1] Python* の表現力とコーディングの容易さは、開発者にとってプロトタイプ環境として魅力的です (**図 2**)。[2,3]

Python* を使用することで、クオンツアナリストによる取引アルゴリズムの開発、データ・サイエンティストによる解析モデルの構築、研究者による数値シミュレーションのプロトタイプ生成が可能になります。



1 2016 年 2 月現在、最も人気があるコーディング言語 (出典: **www.codeeval.com** (英語))



2 プログラミング言語の生産性

Python* パフォーマンスの問題

Python* のパフォーマンスとスケーラビリティーの問題により、通常プロトタイプ・コードをプロダクション・コードにスケーリングするため、開発者は C++ または Java* のような言語でアルゴリズムを書き直す必要があります。例えば、Python* では、評判が悪いグローバル・インタープリター・ロック (GIL) により、効率良い並列コードを記述することができません。アプリケーション・コードの書き直しは、労力と時間がかかるだけでなく、柔軟性を損ない、エラーにつながる可能性があります。

パフォーマンスの問題のほとんどは、Python* インタープリターの実装が現代のハードウェアに対応していないためです。インタープリターは、多種多様なハードウェア・プラットフォームで実行する必要があるため、多くの場合、インテル® ストリーミング SIMD 拡張命令 (インテル® SSE) やインテル® アドバンスト・ベクトル・エクステンション (インテル® AVX) のような CPU 固有の機能を効率良く利用するようにチューニングされていません。

さらに、Python* は、キャッシュの局所性を考慮して実装されていません。そのため、連続するデータがシーケンシャルに格納されず、データアクセスには多くのポインターの逆参照などが必要になります。これらはすべて、Python* インタープリターが最近のハードウェア・プラットフォーム上で優れたパフォーマンスを達成する妨げとなります。

Python* の問題と制限に対する既知のソリューション

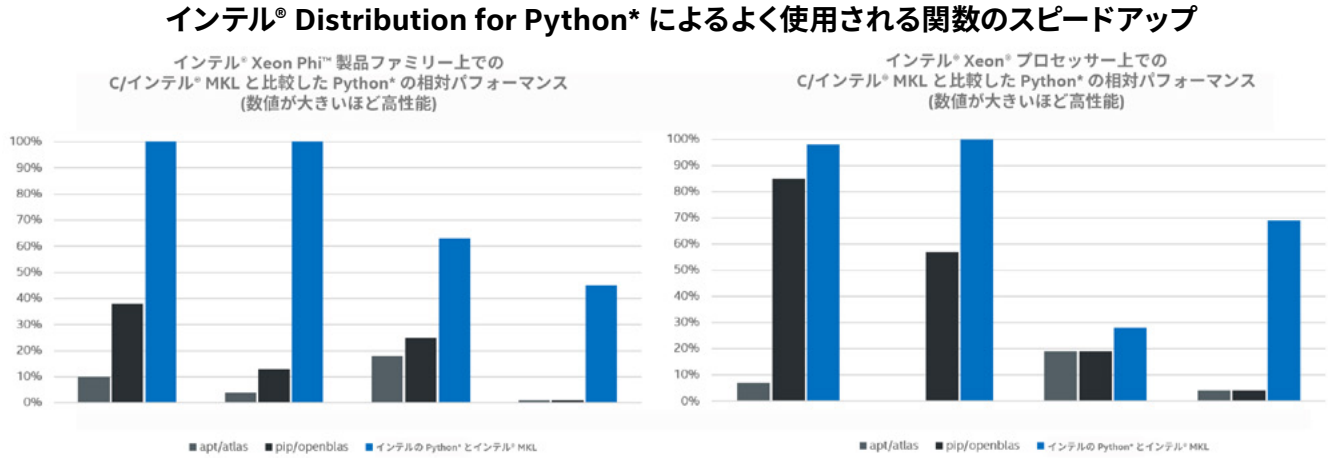
前述のパフォーマンスの問題は通常、Numba のような JIT コンパイラーを使用したり、NumPy*、SciPy*、scikit-learn パッケージの C 拡張を使用して対応できます。これらの Python* パッケージは、ハードウェアを最大限に利用できるように、スレッド対応で高度に最適化されており、パッケージを使用することで、Python* 開発者はアイデアを明確かつ簡潔に表現することができます。

ネイティブにコンパイルされるプログラミング言語を使用する場合、すべての CPU リソースを効率良く利用するのは、一般に開発者の責任です。しかし、多くの Python* 開発者は、アルゴリズムの最適化と並列化に多くの知識を持ち合わせていません (そして、それらの知識の習得は不要であるべきです)。そのため、ライブラリーを通してこれらの最適化を利用するのが簡単です。しかし、NumPy*、SciPy*、scikit-learn のようなライブラリーの標準実装は、汎用化されているため、通常対象とするハードウェアの機能を最大限に引き出せません。例えば、最近のベクトル命令をすべて使用することができません。[4]

Python* コミュニティーへのインテルの貢献

インテルは、Python* のパフォーマンスの課題への取り組みとして、多くのツールと手法を通じて、Python* アプリケーションのパフォーマンスを高速化する、強力で簡単に使えるソリューションを開発者に提供しています。インテル® マス・カーネル・ライブラリー (**インテル® MKL**)、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)、OpenMP* 標準の実装、MPI 標準の実装、インテル® Data Analytics Acceleration Library (インテル® DAAL) を含む、ハイパフォーマンス・アプリケーションの開発を支援する数々のライブラリーに加えて、インテル® VTune™ Amplifer XE などのプロファイリング・ツールがあります。しかし、最近まで、これらの言語とツールのほとんどは、C/C++/Fortran でのみ利用することができました。

インテルは、インテル® MKL を利用して NumPy* のようなパッケージの数値計算を最適化した、**インテル® Distribution for Python*** で Python* 開発者にこれらの強力なツールと手法の提供を開始しました。また、効率良いタスク・スケジューラーによりスレッド処理の調整を行うマルチスレッド・アプリケーションを可能にする Python* 向けのインテル® TBB モジュール [5] と、Python* でのビッグデータ解析を支援する Python* 向けのインテル® DAAL モジュール (pyDAAL) もリリースされました。人気のパフォーマンス・アナライザー **インテル® VTune™ Amplifer XE** では、Python*/C/C++/Fortran が混在するアプリケーションのプロファイルがサポートされ、アプリケーションのパフォーマンスを検証し、自明 / 非自明の対応可能なボトルネックを特定できるようになりました。



ハードウェア/問題サイズ	dot	lu	det	inv	cholesky	fft
インテル® Xeon® プロセッサー (32 コア) および インテル® Xeon Phi™ プロセッサー (64 コア)	(20k, 10k)、(10k, 20k)	(35k, 35k)	(15k, 15k)	(25k, 25k)	(40k, 40k)	520k
インテル® Xeon® プロセッサー (1 コア)	(20k, 5k)、(5,20k)	(20k, 20k)		(10k, 10k)	(10k, 10k)	
インテル® Xeon Phi™ プロセッサー (1 コア)	(20k, 300)、(300, 20k)	(6k, 6k)	(4k, 4k)	(2k, 2k)		

システム構成: システム構成: APT/ATLAS: apt-get でインストール、Ubuntu* 16.10、Python* 3.5.2、NumPy* 1.11.0、SciPy* 0.17.0。pip/OpenBLAS: pip でインストール、Ubuntu* 16.10、Python* 3.5.2、NumPy* 1.11.1、SciPy* 0.18.0。インテルの Python*: インテル® Distribution for Python* 2017。ハードウェア: インテル® Xeon プロセッサー・ベースのシステム: インテル® Xeon® プロセッサー E5-2698 v3 @ 2.30GHz (2 ソケット、1 ソケットあたり 16 コア、HT 無効)、64GB RAM、8 DIMMS (8GB @ 2133MHz)。インテル® Xeon Phi™ プロセッサー・ベースのシステム: インテル® Xeon Phi™ プロセッサー 7210 1.30GHz、96GB RAM、6 DIMMS (16GB @ 1200MHz)。

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。
* 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。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

3 インテル® Distribution for Python* 2017 の Python* パフォーマンス

実際の例へのライブラリーとツールの適応

レコメンデーション・エンジンの構築のような一般的な問題について考えてみます。入力データは、購入商品とその商品に対するユーザーの評価と仮定します。構築されるレコメンデーション・システムは、次の 2 つの独立したタスクを行います : 1) レコメンデーション・ルールの構築、2) 特定のユーザーに対するレコメンデーションの計算。タスク 2 では、システムは Web サービスとして動作し、一度に複数のユーザーからのレコメンデーション要求が発生する可能性があります。

レコメンデーション・ルールの構築

レコメンデーション・システムでモデルの作成に使用されるアルゴリズムは数種類あります。ここでは、商品ベースのレコメンデーションに注目します。モデルは、類似商品の行列で表されます。行列 s_{ij} の各要素は、商品 **i** と **j** の類似性を表します。この実装では、類似性の評価にコサイン距離を使用します。

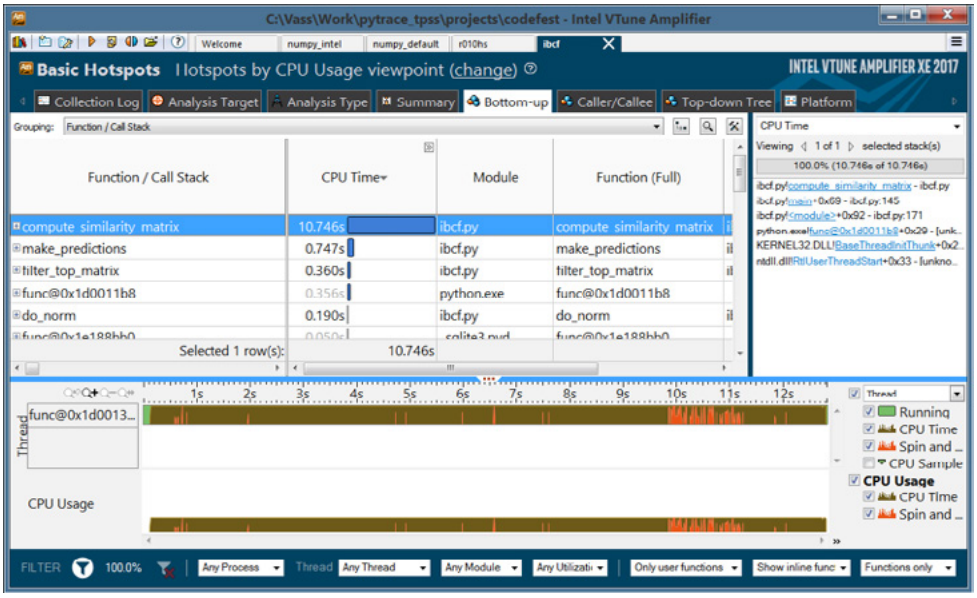
モデル生成アルゴリズムは、次のステップで構成されます。

1. ユーザーの評価をデータベースから行列にロードします。行列の行は個々の商品を、列は個々のユーザーを表します。
2. コサイン距離を使用して商品の類似性を計算します。
   a) 各行をそのユークリッド距離で割って、ユーザーの評価で行列を正規化します。
   b) 行列の行のドット積を計算します。
3. 各商品で、**i** は最も類似性の高い商品を **k** 個選択し、それ以外のすべての商品の類似性をゼロに設定します。

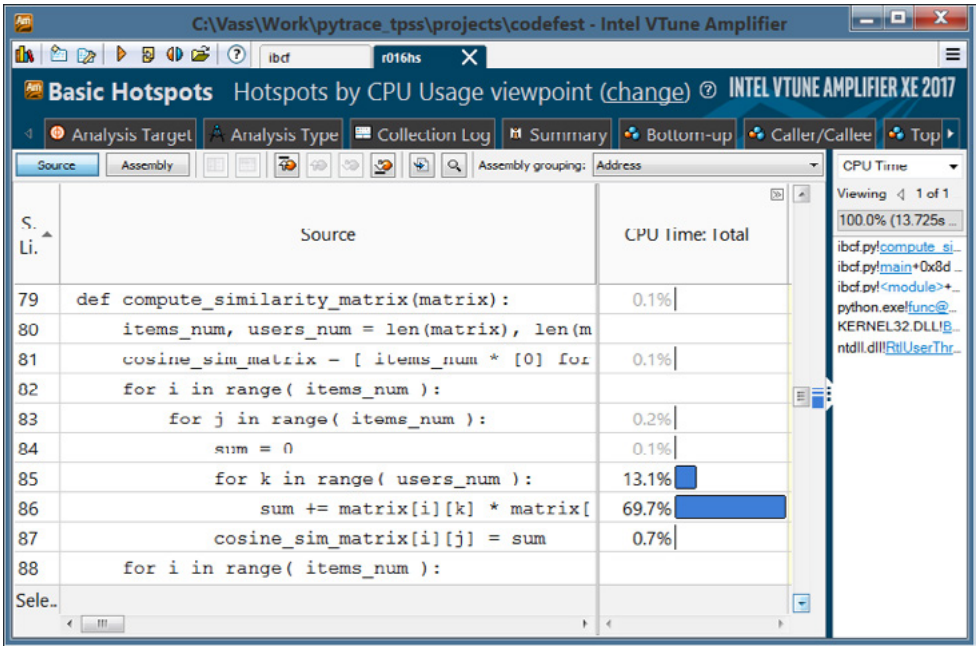
以下は、追加のパッケージを使用せずに、標準の Python* だけでステップ 2b を実装するコードです。

```
def compute_similarity_matrix(matrix):
items_num, users_num = len(matrix), len(matrix[0])
cosine_sim_matrix = [ items_num * [0] for i in range( items_num ) ]
for i in range( items_num ):
for j in range( items_num ):
sum = 0
for k in range( users_num ):
sum += matrix[i][k] * matrix[j][k]
cosine_sim_matrix[i][j] = sum
for i in range( items_num ):
cosine_sim_matrix[i][i] = 0
return cosine_sim_matrix
```

grouplens.org (英語) から、6,040 人のユーザーが 3,260 本の映画に対して行った 1,000,000 件の評価データを使用してこの実装のパフォーマンスを評価したところ、標準の Python* 実装では、データをロードし、類似性行列を作成するのに 338 分かかりました。また、主なボトルネックを理解するため、インテル® VTune™ Amplifer XE の Python* プロファイル機能を利用しました。その際、プロファイル時間を大幅に短縮するため、評価データの件数を 1,000,000 件から 20,000 件に減らしました。**図 4** と**図 5** にプロファイル結果を示します。



4 インテル® VTune™ Amplifer XE の [Bottom-up] ビューから計算負荷が最も高いのは商品の類似性計算であることが分かる

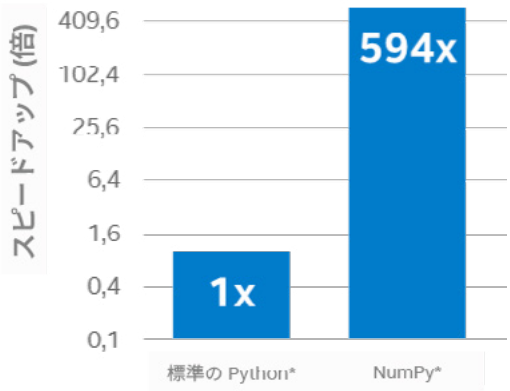


5 インテル® VTune™ Amplifer XE から実行時間の大半が 86 行目で費やされていることが分かる

プロファイル結果から、アルゴリズムで最も実行時間を費やしているのはドット積の計算であることが分かりました。Python* は、この処理を C/C++ や Fortran などのコンパイル言語のように効率良く行うことができません。これは、前述の理由によるものです。幸い、Python* には、追加パッケージとして多数の拡張が用意されており、それらを利用することで、ここで触れたほとんどの問題に対応することができます。NumPy* は、線形代数計算用の業界標準パッケージで、ndarray と呼ばれる追加のデータ型を提供します。ndarray は、連続するメモリーブロックとして表される任意の次元の配列です。NumPy* では、OpenBLAS、Atlas、Netlib のような一般的な数学ライブラリーを利用して線形代数処理を実装しています。

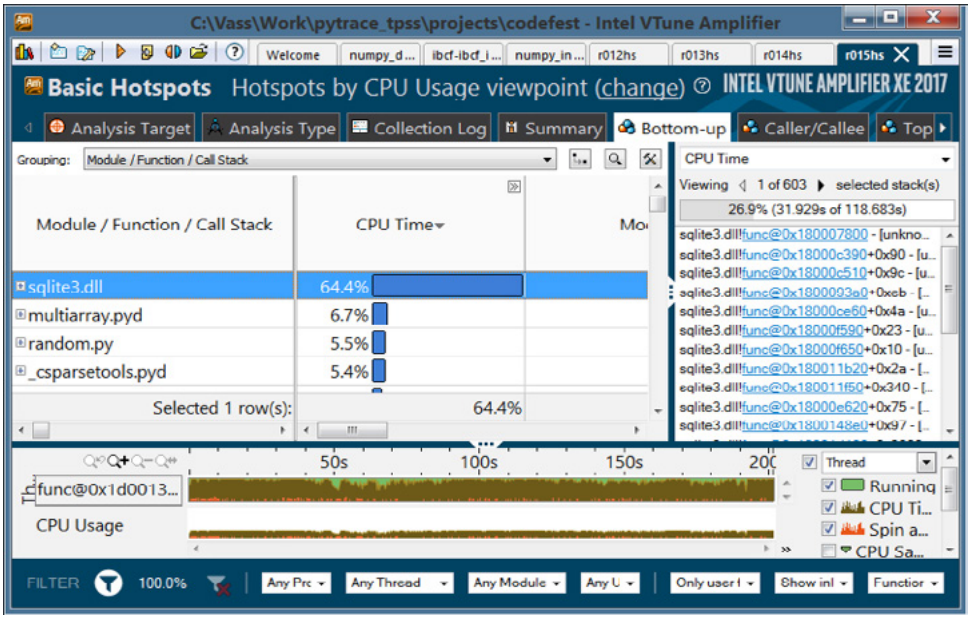
図 6 は、最初の類似性行列計算を次のように変更した後のパフォーマンスの向上を示します。

```
import numpy as np
matrix = np.array(shape=(len(items), len(users)))
...
cosine_sim_matrix = matrix.dot(matrix.T)
```



システム構成 :
Fedora* に同梱の Python*: Python* 2.7.10 (default, Sep 8 2015)、NumPy* 1.9.2、SciPy* 0.14.1、gcc 5.1.1 でビルドされたマルチプロセシング 0.70a1。ハードウェア : 96 CPU (HT 有効)、4 ソケット (1 ソケットあたり 12 コア)、1 NUMA ノード、インテル® Xeon® プロセッサー E5-4657L v2 @ 2.40GHz、RAM 64GB、オペレーティング・システム : Fedora* 23 (twenty-three)。

6 標準の Python* 実装と比較した NumPy* 計算のスピードアップ



7 インテル® VTune™ Amplifer XE の [Bottom-up] ビューからアプリケーションの実行時間の 64.4% がデータベースからのデータのロードに費やされていることが分かる

図 7 のプロファイルから、データのロードが新しいボトルネックであることが分かります。つまり、これ以上計算部分を最適化しても意味がありません。

ユーザーに対するレコメンデーションの生成

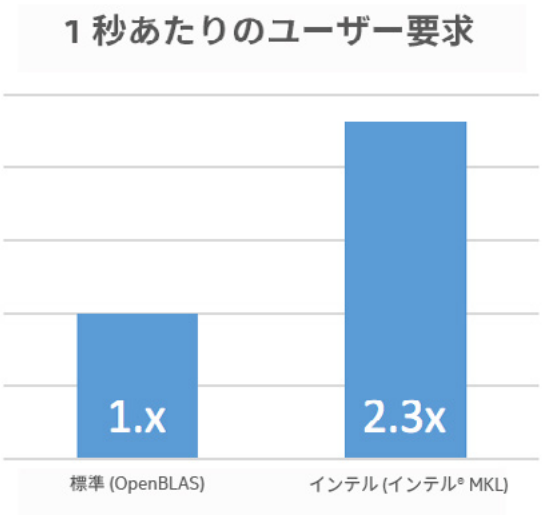
協調フィルタリングの 2 つ目のタスクは、ユーザーのこれまでの評価データを基に未評価商品に対する好みを予測します。商品 **i** のユーザーの好みを計算するため、商品 - 商品の類似性行列の **i** 行目とユーザー評価のベクトルの乗算を行います。類似性行列の各行に対してこの処理を行うことで、すべての商品に対するユーザーの好みを予測します。そして、ユーザーがすでに評価済みの商品を除外後、好みベクトルをソートして、上位 **k** 個の要素を選択します。ここでは、タスクを簡略化するため、上位 1 つのレコメンデーションのみ取得しています。以下のコードは、NumPy* と Numba を使用して、アルゴリズムのこの部分を実装します。

```
@numba.guvectorize('(f8[:],f8[:])', '(),()', target="parallel")
def masking(x, u):
    if u[0]:
       x[0] = 0

x = topk_matrix.dot(user_vector)
masking(x, user_vector)
recommendation_ids = x.argmax(axis=0)
```

関数マスクをネイティブコードにコンパイルし、引数で渡された行列の各要素に並列に適用して、Python* コードを最適化するため、ここでは Numba デコレーターを使用しています。

Web サービスとして動作するアプリケーションでは、レコメンデーションを得るために毎秒数十万の要求を処理する必要がありますが、類似性行列を計算するタスクは、頻繁に実行する必要はありません (例えば、1 日 1 回で十分です)。特定のユーザーに対するレコメンデーション生成タスクは、いくつかの要求をバッチ処理する、大きなタスクのサブセットとして見ることができます。この状況をテストするため、500,000 ユーザーに対するレコメンデーションを一度に計算します。`user_vector` 行列の各次元は、個々のユーザーの評価を表します。

図 8 は、インテル® Xeon Phi™ プロセッサー・ベースのシステム上で、インテル® Distribution for Python* と標準の NumPy* パッケージを使用してレコメンデーションを生成した場合のパフォーマンスです。



システム構成 :
ハードウェア : インテル® Xeon Phi™ プロセッサー 7250、68 コア @1.40GHz、96GB DDR4-2400

バージョン :
インテル® Distribution for Python* 2017 Gold、インテル® MKL バージョン 2017.0.0、libopenblasp-r0-39a31c03.2.18.so、Python* 3.5.2、NumPy* 1.11.1、SciPy* 0.18.0、Red Hat* Enterprise Linux* Server 7.2 (Maipo)

8 標準の Python* + NumPy* パッケージ (OpenBLAS を使用) とインテル® Distribution for Python* (インテル® Xeon Phi™ プロセッサー・ベースのシステム上でインテル® MKL を使用) のユーザーレコメンデーション生成パフォーマンスの比較

標準の NumPy* パッケージは最近のリリースで、インテル® Xeon® プロセッサー上でより高速に実行する OpenBLAS 数学ライブラリーを使用するように変更されました。これにより、標準の NumPy* パッケージの以前のバージョン (OpenBLAS を使用しない) とインテル® Distribution for Python* の間にあったパフォーマンス・ギャップが小さくなりました。標準の NumPy* の新しいバージョンとインテル® Distribution for Python* は、どちらも並列化とベクトル化を最適化しています。しかし、これだけでは、インテル® Xeon Phi™ プロセッサー・ファミリーの新しいインテル® AVX-512 命令と、通常のプロセッサー・アーキテクチャーとは大きく異なるインテル® メニー・インテグレーテッド・コア (インテル® MIC) アーキテクチャーの性能を引き出すことができません。

インテル® Xeon® プロセッサー・ベースのシステムでは、処理をさらにスピードアップできる可能性がもう 1 つあります。ユーザーの好みの計算をアプリケーション・レベルでマルチスレッド化することです。ここでは、NumPy* に似たインターフェイスを備え、NumPy* を呼び出す並列タスクにワークを分割できる Dask ライブラリーを使用します。

Dask を使用するには、例えば、NumPy* 配列の代わりに Dask 配列を宣言する必要があります。

```
chunks=(int(topk_matrix.shape[0]), 5000) # 1 タスクあたり 5000 ユーザー
t = dask.array.from_array(topk_matrix, chunks)
u = dask.array.from_array(user_vector, chunks)
```

この例では、以前に宣言した行列をソースデータとして使用していますが、実際のアプリケーションは最初から Dask 配列を扱うことができるため、変換オーバーヘッドは発生しません。上記の NumPy* コードは、次のように変更できます。変更後のコードは、上記のコードで宣言されたものと同じ **`masking`** 関数を使用します。

```
x = t.dot(u)
dask.array.map_blocks(masking, x, u)
recommendation_ids = x.argmax(axis=0).compute()
```

このコードは、複数のスレッドを使用して、配列の異なるチャンクを並列に処理するタスクを生成します。実行時間の大半がグローバル・インタープリター・ロック (GIL) を解放する NumPy* 内で費やされているため、GIL は大きな問題にはなりません。そのため、複数の計算を並列に実行できます。しかし、NumPy* 計算の並列実行は、オーバーサブスクリプションと呼ばれる別の問題を引き起こす可能性があります。[6] この問題は、2 つの入れ子構造レベル (1 つは Dask から、もう 1 つは**インテル® MKL** 内) で同時にワークが並列実行されると発生します。どちらの並列レベルも、システム全体がアイドル状態で、直ちに並列処理を行うことができるかのように振る舞います。最悪の場合、作成されるスレッド数はコア数の 2 乗になり、アプリケーションは有用なワークの代わりに、コンテキスト・スイッチやキャッシュミスに大半の時間を費やします。オーバーサブスクリプションの問題に対応するため、ここでは優れた入れ子構造の並列処理を提供するインテル® TBB ライブラリーを使用します。インテル® TBB の Python* ラッパーは、Dask で使用される標準の Python* の ThreadPool クラスを置換できます。Dask やアプリケーションのソースコードを変更する必要はありません。次のコマンドライン・オプションを指定すると、インテル® TBB の Python* ラッパーが有効になります。

```
$ python -m TBB <your>.py
```

このオプションは、インテル® MKL でインテル® TBB のスレッドレイヤーも有効にするため、両方の並列レベルで、インテル® TBB のタスク・スケジューラーによってタスクと利用可能な処理ユニットの調整が行われます。

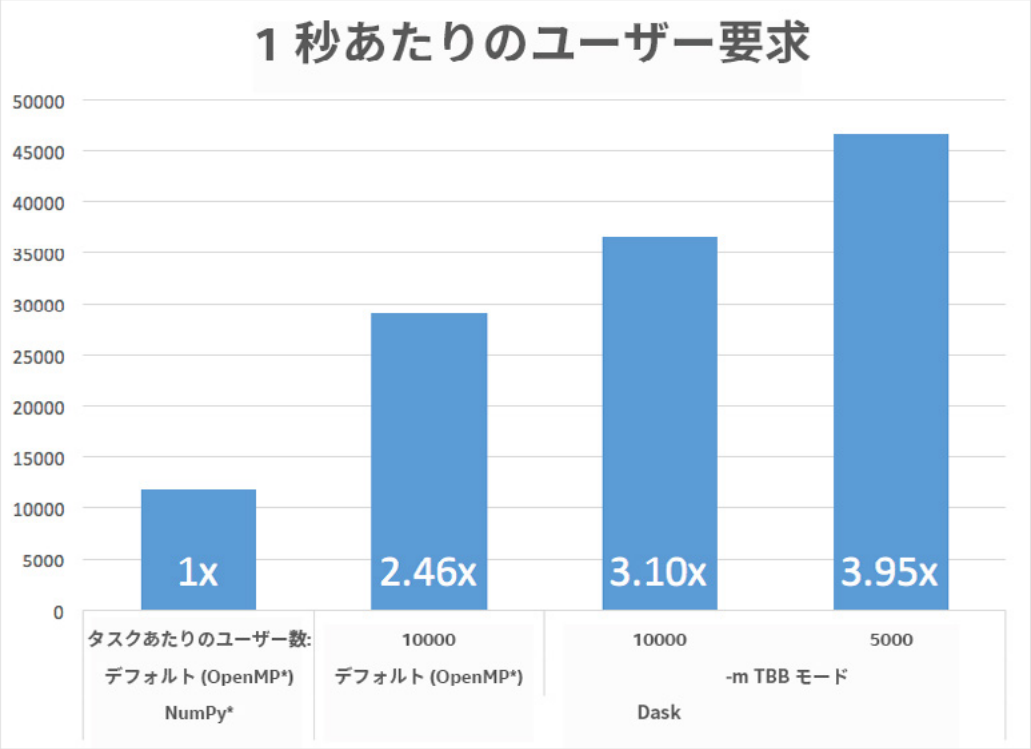
オーバーサブスクリプションは、パフォーマンスに影響するだけではありません。多数のプロセッサー・コアが搭載された大規模システムでは、オペレーティング・システムが多数のスレッドの作成を拒否し、次のようなメッセージを出力します。

```
OMP: エラー #34: システムは OMP スレッドに必要なリソースを割り当てることができません:
OMP: システムエラー #11: Resource temporarily unavailable
OMP: ヒント: OMP_NUM_THREADS の値を減らしてみてください。
```

(OpenBLAS で高速化された) 標準の NumPy* を実行しているシステムとアプリケーションは、追加の情報やヒントを表示することなく失敗します。セグメンテーション違反が発生する可能性もあります。インテル® TBB の Python* モジュールは、この問題が発生した場合にアプリケーション側で対応するための簡単な方法を提供します。OMP_NUM_THREADS=1 を設定してプログラムを続行することもできますが、プロセス全体で並列処理が制限されます。インテル® TBB モードで実行中、プログラムは動的にスケーリングし、バランスを取ります。

図 9 は、上記のコードベースのベンチマーク・アプリケーションを実行した結果です。

デフォルトの NumPy* バージョンは、どの Dask (マルチスレッド) バージョンよりも遅くなります。デフォルト (OpenMP*) モードの Dask バージョンは、1 タスクあたり 10,000 ユーザーでのみ実行できました。1 タスクあたり 5,000 では、常に前述の “OMP: エラー #34” で失敗しました。最良の結果は、インテル® TBB モードで 1 タスクあたり 5,000 ユーザーのチャンク (合計 500,000 ユーザー) を実行した際に得られました。



システム構成 :
ハードウェア : インテル® Xeon® プロセッサー E7-8890 v4、4 x 24 コア @ 2.20GHz (最大 4GHz)、HT 無効、768GB DDR4。
バージョン : インテル® Distribution for Python* 2017 Gold、インテル® MKL バージョン 2017.0.0、Python* 3.5.2、NumPy* 1.11.1、SciPy* 0.18.0、Numba 0.26.0、llvmlite 0.11.0、Dask 0.11.0、CentOS* Linux* 7.2.1511 (Core)。

9 異なるパラメーター設定による NumPy* と Dask バージョンのレコメンデーション生成ベンチマークのパフォーマンスの比較

分散プログラミング

実際のアプリケーションからのデータは膨大な量に上ることがあります。Netflix* のコンテストでは、480,000 メンバーの 18,000 本の映画の評価が行列に格納され、行列サイズは 65GB でした。この膨大な量の計算とデータ割り当てには、協調フィルタリング・アルゴリズムの分散バージョンを使用できます。

インテル® DAAL は、データの前処理からデータマイニングおよびマシンラーニングまで、すべてのデータ解析段階に対応したアルゴリズムを提供します。テキストファイル、データベース、分散ファイルシステムを含む外部データソースからデータをロードし、実際の計算を行うまでの計算フロー全体を最適化するように設計されています。インテル® DAAL は C++ で記述されていますが、インテル® Distribution for Python* の pyDAAL コンポーネントで Python* ラッパーが用意されています。

インテル® DAAL は、いくつかの処理モードをサポートします。

- **バッチ :** このモードは、すべてのデータが 1 つのデバイス上にあり、一度に処理可能な場合に使用します。
- **オンライン :** このモードは、データをブロック単位で受け取る場合、またはデータが大きすぎて一度にデバイスのメモリーに収まらない場合に使用します。この場合、部分結果を段階的に更新して、データをデバイスメモリーにストリームし、次のデータブロックを処理する前にファイナライズします。
- **分散 :** このモードは、データセットが大きすぎて 1 つのデバイスのメモリーに収まらない場合、または複数のデバイスで実行することでアルゴリズムの処理時間を短縮できる場合に使用します。この場合、アルゴリズムは各ノードで部分結果を生成し、最後にマスターノードで最終結果にマージされます。

それぞれの処理モードで使用される API は似ており、必要に応じて、あるモードから別のモードへ比較的簡単に切り替えることができます。例えば、プロトタイプの生成段階ではバッチ処理モードと比較的小さなデータセットを使用し、プロダクション段階では分散処理モードと大きなデータセットに切り替えることができます。

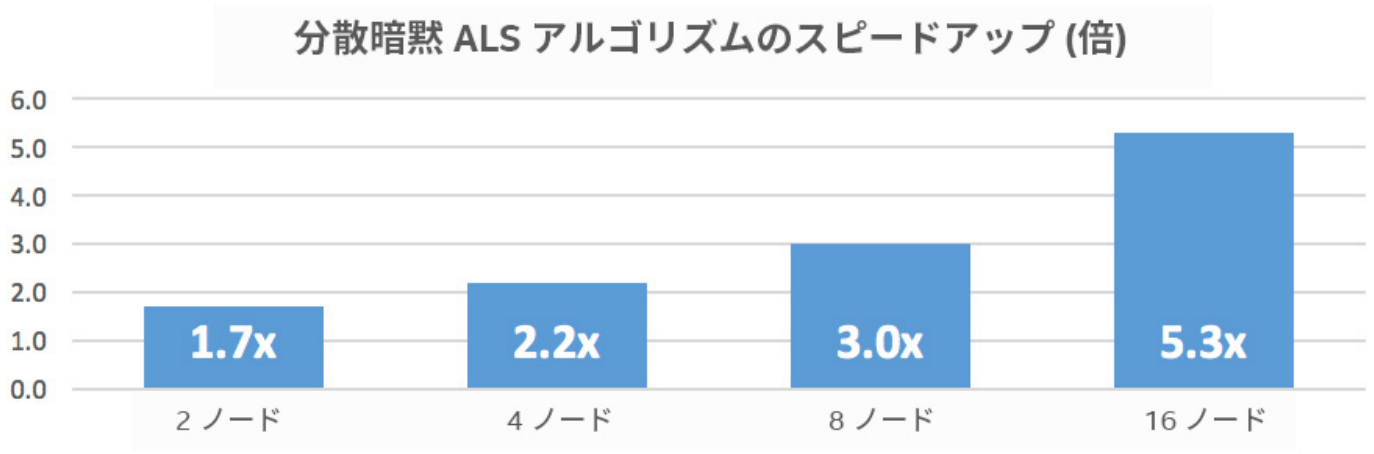
インテル® DAAL の分散処理モードは、通信レイヤーに依存せず、mpi4py や PySpark とともに利用できます。特定のハードウェア向けに最適な通信スキームを選択するのはユーザーの責任ですが、インテルのパッケージには、さまざまな通信技術を利用して分散アルゴリズムを実行する方法を示すサンプルが含まれています。

インテル® DAAL は、バッチおよび分散処理モードの協調フィルタリング向けに暗黙の交互最小 2 乗 (暗黙 ALS) アルゴリズムを提供します。mpi4py と pyDAAL を利用して分散協調フィルタリングを実装する方法を示すソースコードは、**インテル® DAAL のサンプルコード・ページ**から入手できます。

インテル® DAAL の暗黙 ALS のパフォーマンスを示すため、次のベンチマークを用意しました。このベンチマークは、分散暗黙 ALS アルゴリズムを、人為的に生成された 10,000,000 ユーザーの 10,000,000 商品に対する評価を含むデータセットに適用します。データセットはスパースで、すべてのユーザーと商品に一様に分布された 100,000,000 件の既存の評価が含まれており、それ以外の評価はゼロに設定されています。インテル® DAAL の実装は、CSR (Compressed Sparse Row) 形式のスパースデータを扱うため、ベンチマーク・データはこの形式で格納されています。暗黙 ALS アルゴリズムのパラメーターのうち、計算の複雑さに大きく影響する因数の数と反復数は、ぞれぞれ 10 と 15 に設定されています。

このベンチマークは、通信レイヤーに **MPI** を使用し、1 から 16 ノードの範囲でスケーリングします。各ノードで 1 つの MPI ランクが開始されます。データセットは、ノード数に応じていくつかのパートに分割されます。

図 10 は、それぞれのクラスターノード数での暗黙 ALS トレーニング・アルゴリズムのパフォーマンス・スケーリングを示したものです。シングルノードでは、計算に 192.8 秒かかります。グラフでは、この値をベースに相対的なパフォーマンスの向上を示しています。



システム構成:
ハードウェア (各ノード): インテル® Xeon® プロセッサー E5-2697 v4 @ 2.30GHz、2 x 18 コア、HT 有効、128GB RAM。
バージョン: Oracle* Linux* Server 6.6、インテル® DAAL 2017 Gold、インテル® MPI ライブラリー 5.1.3。
インターコネクト: 1GB Ethernet。

10 異なるクラスターノード数での分散暗黙 ALS アルゴリズムのパフォーマンス・スケーリング

まとめ

複数のツールと手法により、Python* アプリケーションのパフォーマンスを大幅に向上し、ネイティブコードのスピードに近づけることができます。協調フィルタリング・アプリケーションの例から、インテル® Distribution for Python* とインテル® VTune™ Amplifer XE に含まれる Python* ツールは、インテル® プロセッサー上で有用であることが分かります。標準の Python* アプリケーションのプロファイルでは、ドット積関数でパフォーマンス・ボトルネックが正確に検出されます。

インテル® Distribution for Python* に含まれるインテル® MKL により最適化された NumPy* を使用することで、インテル® Xeon Phi™ プロセッサー・ベースのシステム上で、標準の NumPy* バージョンよりもパフォーマンスを 2.3 倍向上できます。

スレッドのオーバーサブスクリプションの問題は、インテル® TBB パッケージにより対応できます。効率良いスレッド・スケジューリングによって入れ子構造の並列処理のパフォーマンスを向上し、シングルスレッド・アプリケーションと比べて最大 4 倍のスピードアップを達成できます。

インテル® DAAL の暗黙 ALS アルゴリズムは、パフォーマンスを高速化するため、インテル® Xeon® プロセッサー・ベースのノード上で分散処理モードで実行する大規模データセット向けに実装されています。

Python* は、誰もがコンピューティングを利用できるようにしました。Python* 向けの強力なパフォーマンス・ツールは、開発者とユーザーが、パフォーマンスを諦めることなく、Python* を使用できるようにします。

インテル® DAAL の詳細 >

インテル® DISTRIBUTION FOR PYTHON* の詳細 >

参考資料 (英語)

1. Studies on popularity of coding languages by **www.codeeval.com**, February 2016, **blog.codeeval.com/codeevalblog/2016/2/2/most-popular-coding-languages-of-2016**.
2. An empirical comparison of seven programming languages, L. Prechelt, IEEE Computer, 2000, Vol. 33, Issue 10, pp. 23–29.
3. Programming languages ranked by expressiveness, D. Berkholz, RedMonk, March 2013, **redmonk.com/dberkholz/2013/03/25/programming-languages-ranked-by-expressiveness**.
4. HPC Trends and What They Mean for Me, J. Cownie, Imperial College, Oct. 2015.
5. Unleash parallel performance of Python programs, A. Malakhov, April 2016, **software.intel.com/en-us/blogs/2016/04/04/unleash-parallel-performance-of-python-programs**.
6. Collaborative Filtering for Implicit Feedback Datasets, Y. Hu, Y. Koren, and C. Volinsky, Eighth IEEE International Conference on Data Mining, Pisa, 2008, pp. 263–272.

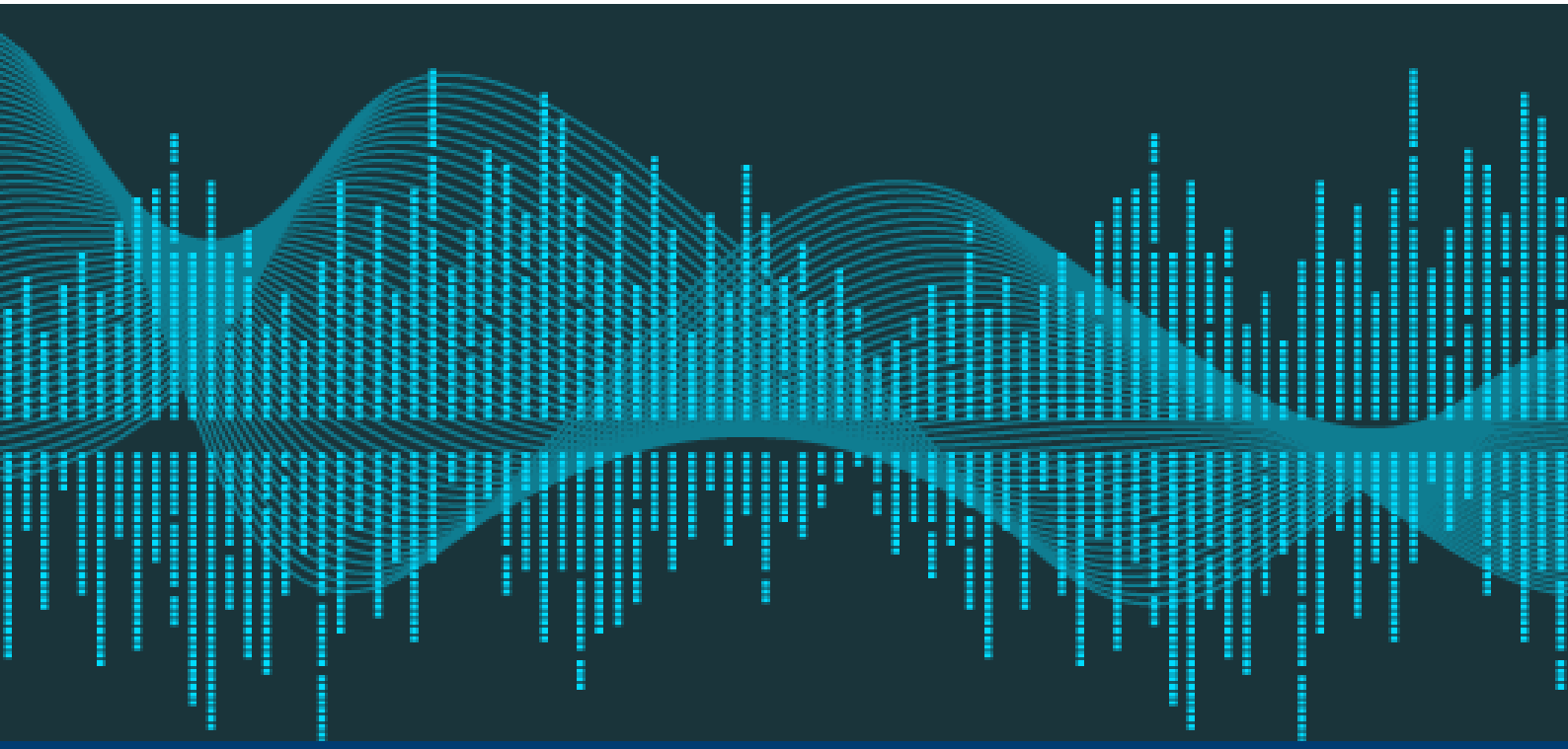
BLOG HIGHLIGHTS

Wind River* Simics で Simics を実行してカーネル 1-2-3 バグを発見

JAKOB ENGBLOM >

私は、バグ、ソフトウェアの不可解な / 予想外の動作、デバッグに関する話が大好きです。これらは、ソフトウェア開発というドライな世界に、笑いと騒動をもたらします。バグによっては、解決してみれば大したことがないものであったり、人に話すのが恥ずかしいものもあります。あるいは、伝説のイエティのように捉えどころがなく、見つからないものもあります。さらに、勇敢なプログラマーが素晴らしいひらめきと粘り強さにより、ドラゴン (デバッグ) を制した英雄的物語のようなものもあります。これはそんなバグの 1 つで、Linux* カーネルで見つけた非常に困難なバグに関する話です。Wind River* Simics (Simics) で Simics を実行して見つけました。このバグを再現するには、1 つのネットワーク、2 台のマシン、少なくとも 3 つのプロセッサー・コア (タイトルの 1-2-3 はこれに由来します) が必要です。このバグは、シングル・プロセッサー上でマルチタスクだけでなく、コンカレンシーを必要とします。そして、バグを見つけるため、ユーザー空間と Linux* カーネルコードの両方を延々とたどっていく必要があります。

この記事の続きはこちら (英語) でご覧になれます。>



インテル® VTUNE™ AMPLIFIER XE による JAVA* および PYTHON* コードのプロファイル

Java* および Python* ベースのアプリケーションで CPU 性能を引き出す

Sukruv Hv インテル コーポレーション ソフトウェア・テクニカル・コンサルティング・エンジニア

長年にわたり、Java* は開発者に人気のある言語で、エンタープライズ、組込み、IoT (モノのインターネット) 向けアプリケーションを牽引してきました。インテル® VTune™ Amplifier XE の電力およびパフォーマンス・プロファイラー (インテル® ソフトウェア・ツール・スイートに含まれる) は、長い間、Java* や .NET のようなマネージドコードのプロファイルを行ってきました。Python* も、使いやすさとシステムを効率良く統合できることから、人気のプログラミング言語になりました。

この 2 つの異なるプログラミング言語の人気の高まりを受け、これらの言語で開発されたアプリケーションが CPU 性能を効率良く利用できるように、インテル® VTune™ Amplifier XE のプロファイル機能が拡張され、Java* および Python* ベースのアプリケーションに対応しました。ここでは、インテル® VTune™ Amplifier XE を使用して、アプリケーションの振る舞いを詳しく理解します。

インテル® VTune™ Amplifier XE とは ?

インテル® VTune™ Amplifier XE は、多くの CPU 時間を費やしているモジュール / プロセス (hotspot) の特定を支援する、パフォーマンスおよび電力プロファイラーです。サンプリングにより、最小限のオーバーヘッドで、マイクロアーキテクチャー・レベルの問題 (例えば、キャッシュミス、ページウォーク、TLB 問題など) を見つけることができます。

アプリケーション・パフォーマンスをプロファイルする理由

アプリケーション・パフォーマンスのプロファイルは、多くの CPU クロックサイクルを費やしているさまざまなコードブロックを特定するのに必要です。この情報は、手動でタイミング API を挿入して取得することもできます。ただし、プロジェクトに多数のモジュールが含まれる場合、手動では問題のあるモジュールの検証と特定に多くの時間がかかります。インテル® VTune™ Amplifier XE の各種プロファイラーは、問題にドリルダウンするのに役立ちます。

インテル® VTune™ Amplifier XE の機能

インテル® VTune™ Amplifier XE には、直感的で使いやすいさまざまな機能があります。

- **サンプリング :** インストルメンテーションと比べて最小限のオーバーヘッドでプロファイルできます。
- **使いやすいプロファイラー :** さまざまなグループ / フィルター / 呼び出し元 - 呼び出し先オプションにより、問題のコード領域を絞り込むことができます。
- **異なるレベルの情報 :** ソースレベルとアセンブリー・レベルで情報を提供します。
- **マイクロアーキテクチャーに関する情報 :** アプリケーションのマイクロアーキテクチャーに関するさまざまな情報を提供します。

Java* アプリケーションのプロファイル

Java* アプリケーションのプロファイルは、4 つのステップから成ります。

1. インテル® VTune™ Amplifier XE プロジェクトを作成します。
2. プロファイルするアプリケーション / プロセスを選択します。
3. 解析タイプを選択します。
4. 結果を収集して、解釈します。

Windows® 上でのインテル® VTune™ Amplifier XE プロジェクトの作成

最初に、amplxe-vars バッチファイルを使用して環境変数を設定します。例えば、インテル® VTune™ Amplifier XE がデフォルトのディレクトリーにインストールされている場合は、次のコマンドを実行します。

```
C:\[Program Files]\IntelSWTools\VTune Amplifier XE\amplxe-vars.bat
```

バッチファイルは、製品名とビルド番号を表示します。

次に、インテル® VTune™ Amplifier XE を起動します。スタンドアロン GUI インターフェイスの場合は、**`amplxe-gui`** コマンドを実行します。コマンドライン・インターフェイスの場合は、**`amplxe-cl`** コマンドを実行します。

インテル® VTune™ Amplifier XE プロジェクトを作成します (スタンドアロンの場合のみ)。

- 右上のメニューボタンをクリックして、[New] > [Project] を選択します。
- [Create Project] ダイアログボックスでプロジェクトの名前と場所を指定します。

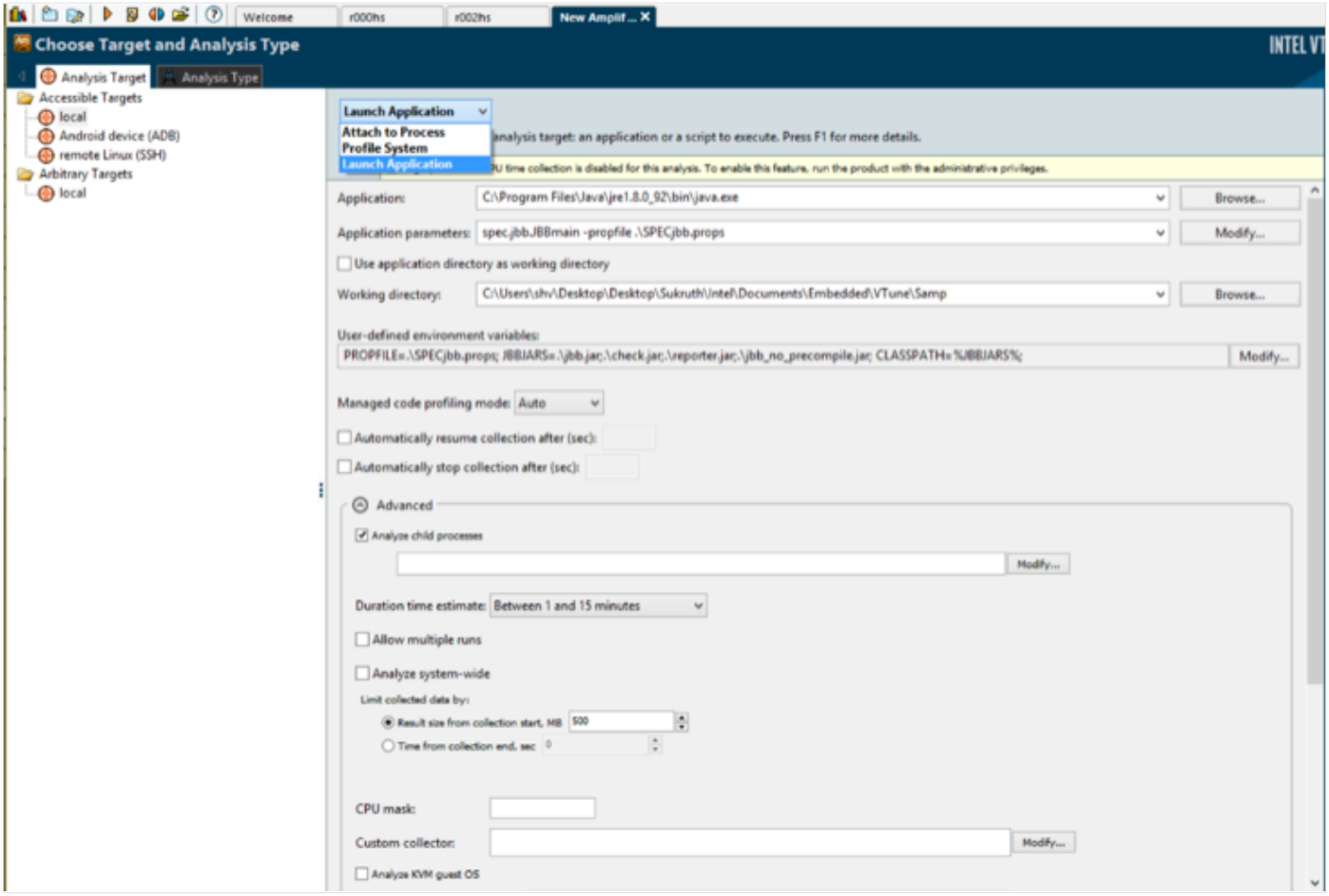
注 : Linux* プラットフォーム上でプロジェクトを新規作成する方法は、**「インテル® VTune™ Amplifier XE 2017 for Linux* 入門」** (英語) を参照してください。

プロファイルするアプリケーション / プロセスの選択

[Analysis Target] タブの左ペインでターゲットシステムを選択し、右ペインで解析ターゲットタイプを選択します。次のいずれかを選択できます。

- **Launch Application:** プロファイルするアプリケーションを起動します。
- **Attach to Process:** すでに実行中のアプリケーションにプロファイラーをアタッチします。
- **Profile System:** アプリケーションとシステムコールの相互作用を検証します。

[Launch Application] を選択する場合、[Application] フィールドに java.exe ファイルのパスを、[Application parameters] に "Java options" や "prop files" のようなパラメーターを指定します (**図 1**)。そして、[User-defined environment variables] フィールドで、アプリケーションの実行に必要な環境変数を指定します。別の方法として、*.bat ファイルで各種環境変数、java.exe バイナリー、パラメーターを定義して、[Application] フィールドで java.exe ファイルの代わりにその *.bat ファイルを指定することもできます。



1 ターゲットと解析タイプの選択

解析タイプの選択

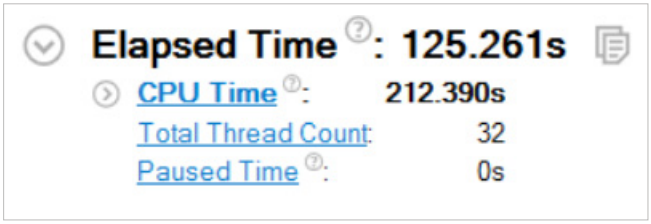
[Analysis Type] タブに移動して、[Basic Hotspots] 解析タイプを選択し、[Start] をクリックします。

結果の収集と解釈

ここでは、SPECjbb2000 ベンチマークを使用しています。

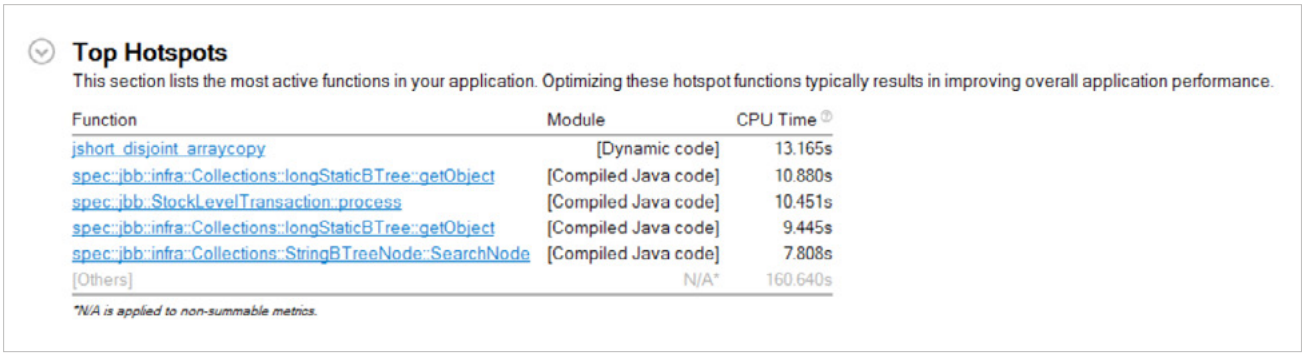
データ収集とファイナライズが完了すると、[Summary] タブが開き、各種メトリックが表示されます。いくつかのメトリックについて見ていきましょう。

- **[Summary] タブの最初のセクション (図 2)** には、アプリケーションの Elapsed Time (経過時間 : アプリケーションの開始から終了までのウォールクロック時間)、CPU Time (CPU 時間 : アプリケーションが CPU を使用していた時間)、Total Thread Count (合計スレッド数 : アプリケーションで使用されたスレッド数) が表示されます。



2 経過時間

- **[Summary] タブの 2 つ目のセクション (図 3)** には、Top Hotspots (上位の hotspot: 多くの CPU 時間を費やしている関数) が表示されます。この例では、5 つの hotspot が見つかり、1 つ目の hotspot は 13.165 秒を費やしています。[Module] 列に表示されている [Dynamic code] と [Compiled Java code] は Java* メソッドで、ネイティブバイナリーのモジュールに関連付けることはできません。すべての Java* ユーザーメソッドは、[Compiled Java code] モジュールに関連付けられます。また、JVM は内部使用目的にいくつかのメソッドを生成し、Java* プロファイラーにレポートしますが、これらは [Dynamic code] モジュールに関連付けられます。



Top Hotspots

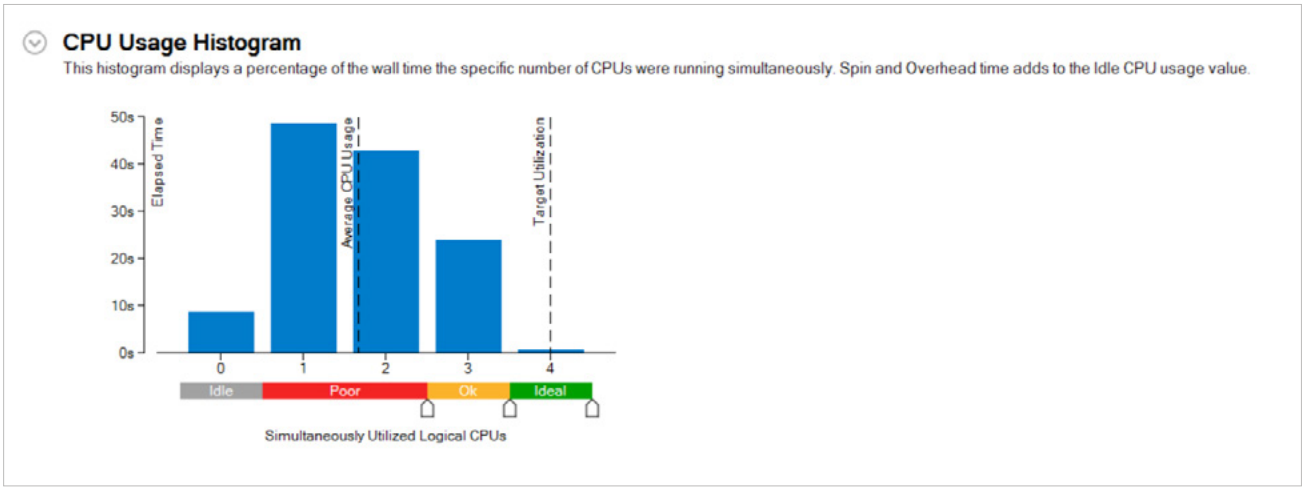
This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time
jshort_disjoint_arraycopy	[Dynamic code]	13.165s
spec::jbb::infra::Collections::longStaticBTree::getObject	[Compiled Java code]	10.880s
spec::jbb::StockLevelTransaction::process	[Compiled Java code]	10.451s
spec::jbb::infra::Collections::longStaticBTree::getObject	[Compiled Java code]	9.445s
spec::jbb::infra::Collections::StringBTreeNode::SearchNode	[Compiled Java code]	7.808s
[Others]	N/A*	160.640s

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

3 上位の hotspot

[Summary] タブの 3 つ目のセクション (図 4) には、CPU Usage Histogram (CPU 使用ヒストグラム : アプリケーションの実行に使用された論理コア数のグラフ) が表示されます。インテル® VTune™ Amplifier XE は、アプリケーションで使用されたコア数を Idle (アイドル)、Poor (低)、OK (許容範囲)、Ideal (理想) に分類し、色分けして表示します。



4 CPU 使用ヒストグラム

[Bottom-up] タブからは、さまざまな詳細が得られます (図 5)。**グループ**や**フィルター**を使用したり、特定の期間に絞って、各スレッドのワークロードをタイムライン表示することができます。関数 / モジュールをダブルクリックすることで、対応するソース行へドリルダウンすることも可能です。

Function / Call Stack	CPU Time: Effective Time by Utilization (Idle, Poor, Ok, Ideal, Over)	Spin Time	Ove... Time	Module	Function (Full)	Sour... File	Start Address
jshort_disjoint_arraycopy	13.165s	0s	0s	[Dynamic code]	jshort_disjoint_ar ...		0x19c22980
spec::jbb::infra::Collections::longStaticBT	10.880s	0s	0s	[Compiled Java code]	spec::jbb::infra::C ...	long ...	0x1a1991ae
spec::jbb::StockLevelTransaction::process	10.451s	0s	0s	[Compiled Java code]	spec::jbb::StockL ...	Stoc...	0x19fb6413
spec::jbb::infra::Collections::longStaticBT	9.445s	0s	0s	[Compiled Java code]	spec::jbb::infra::C ...	long ...	0x1a19931f
spec::jbb::infra::Collections::StringBTreeN	7.808s	0s	0s	[Compiled Java code]	spec::jbb::infra::C ...	Strin...	0x19e08420
spec::jbb::infra::Collections::longStaticBT	5.999s	0s	0s	[Compiled Java code]	spec::jbb::infra::C ...	long ...	0x1a199283
spec::jbb::infra::Collections::longStaticBT	5.988s	0s	0s	[Compiled Java code]	spec::jbb::infra::C ...	long ...	0x1a198da8
java::util::Hashtable::put	3.499s	0s	0s	[Compiled Java code]	java::util::Hashta ...		0x19cf9b20
spec::jbb::Orderline::process	3.188s	0s	0s	[Compiled Java code]	spec::jbb::Orderli ...	Orde...	0x19e256a9
spec::jbb::infra::Util::DisplayScreen::privIn	2.846s	0s	0s	[Compiled Java code]	spec::jbb::infra::U ...	Displ...	0x1a0807ec
spec::jbb::infra::Util::DisplayScreen::putDc	2.804s	0s	0s	[Compiled Java code]	spec::jbb::infra::U ...	Displ...	0x19efa0e0
spec::jbb::infra::Collections::longStaticBT	2.557s	0s	0s	[Compiled Java code]	spec::jbb::infra::C ...	long ...	0x19f40ebf
spec::jbb::infra::Collections::longStaticBT	2.349s	0s	0s	[Compiled Java code]	spec::jbb::infra::C ...	long ...	0x19f41479
spec::jbb::Orderline::validateAndProcess	2.218s	0s	0s	[Compiled Java code]	spec::jbb::Orderli ...	Orde...	0x1a199144
[Unknown stack frame(s)]	2.140s	0s	0s		[Unknown stack ...		0
WriteFile	1.823s	0s	0s	KERNELBASE.dll	WriteFile		0x180001 ...
Selected 1 row(s):	10.880s	0s	0s				

5 [Bottom-up] タブ

コードの最適化への取り掛かりとして、特定のソース行へのドリルダウンが役立ちます。 また、ハイライトしたソース行に対応するアセンブリー・コードも確認することができます (**図 6**)。

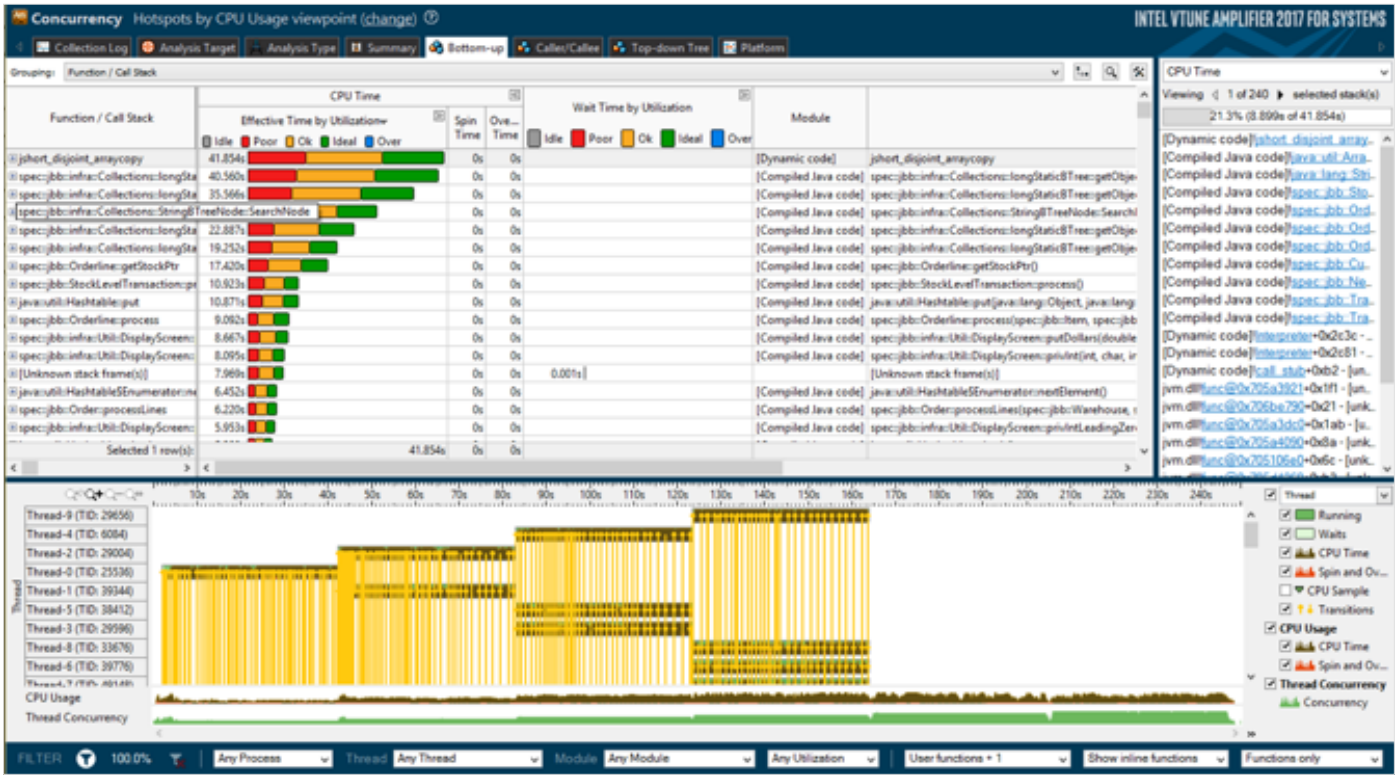
Source | Assembly | Assembly grouping: Address

So. Li.	Source	CPU Ti Effective Time by Ut (Idle, Poor, Ok, Ide)
1,193	System.out.print (m);	
1,194	System.out.print (' ');	
1,195	System.out.print (cnt);	
1,196	System.out.print (' ');	
1,197	System.out.print (target);	
1,198	System.out.print (' ');	
1,199	System.out.print (tkeys[pos]);	
1,200	System.out.println();	
1,201	*/	
1,202	pos--;	
1,203	};	
1,204	} else {	
1,205	//System.out.println(" getObject:: pos :	
1,206	pos = cnt;	
1,207	};	
1,208	};	
1,209		
1,210	if (target != tkeys[pos])	
1,211	searchNode = ((longBTreeNode)searchNode	6.815s
1,212	else {	
1,213	result = searchNode.vals[pos];	
1,214	if (debugPrint)	
1,215	{	
1,216	System.out.print("LongBTree::getObject"	
1,217	System.out.print(" found result=");	
1,218	System.out.print(result);	
1,219	System.out.println();	
1,220	};	
1,221	break;	
1,222	};	
1,223	};	
1,224	} while (searchNode != null);	0.050s
1,225		

Address	Sour... Line	Assembly	CP Effective Time by (Idle, Poor, Ok)
0x1a1991ae		Block 1:	
0x1a1991ae	1,211	mov ecx, dword ptr [r11+0x8]	0.1%
0x1a1991b2	1,211	cmp ecx, 0x2000f1ba	3.1%
0x1a1991b8	1,211	jnz 0x1a19a491	
0x1a1991be		Block 2:	
0x1a1991be	1,224	test dword ptr [rip-0x29091c4], eax	0.0%
0x1a1991c4	1,224	mov rdi, r11	0.0%
0x1a1991c7	1,146	mov ecx, dword ptr [rdi+0x18]	0.0%
0x1a1991ca	1,146	mov r9d, dword ptr [rcx+0xc]	0.2%
0x1a1991ce	1,146	cmp r9d, 0x1	0.4%
0x1a1991d2	1,146	jbe 0x1a199679	
0x1a1991d8		Block 3:	
0x1a1991d8	1,147	mov esi, dword ptr [rdi+0x1c]	0.0%
0x1a1991db	1,148	cmp r10, qword ptr [rcx+0x18]	0.0%
0x1a1991df	1,148	jl 0x1a199425	
0x1a1991e5		Block 4:	
0x1a1991e5	1,147	movsx r11d, word ptr [rdi+0x14]	0.2%
0x1a1991ea	1,150	cmp r11d, 0x10	0.0%
0x1a1991ee	1,150	jnl 0x1a199b01	
0x1a1991f4		Block 5:	
0x1a1991f4	1,152	cmp r11d, 0x1	0.0%
0x1a1991f8	1,152	jle 0x1a198da8	
0x1a1991fe		Block 6:	
0x1a1991fe	1,154	cmp r9d, 0x2	0.0%
0x1a199202	1,154	jbe 0x1a1998c1	
0x1a199208		Block 7:	
0x1a199208	1,154	cmp r11d, r9d	0.0%
0x1a19920b	1,154	jnb 0x1a1998c1	
0x1a199211		Block 8:	
0x1a199211	1,154	mov ebx, r11d	0.0%
0x1a199214	1,154	dec ebx	0.0%
0x1a199216		Block 9:	
0x1a199216	1,154	mov rdx, qword ptr [rcx+r11*8+0x10]	0.0%

6 ソース行に対応するアセンブリー

Basic Hotspots 解析タイプで多くの CPU リソースを消費しているモジュールを把握したら、**Concurrency** 解析タイプを使用して、コンテキスト・スイッチとスレッド間の遷移を確認できます (**図 7**)。黄色の線は、スレッド間の遷移を示します。



7 Concurrency 解析

General Exploration 解析タイプは、**アーキテクチャー解析**を実行して、キャッシュ、TLB、ページウォーク、フォルス・シェアリングなどに関するさまざまな問題の詳細を提供し、これらの問題を理解するのを助けます。

Python* アプリケーションのプロファイル

Python* プロファイルは、インテル® VTune™ Amplifier XE 2017 の新機能です。

インテル® VTune™ Amplifier XE で Python* アプリケーションをプロファイルする方法は、**Parallel Universe 25 号**の「インテル® VTune™ Amplifier XE を利用した Python* コードの高速化」を参照してください。本記事では、混在モードのプロファイル、*.pyd モジュールのシンボルの解決、**コンパイラー**の設定、Cython* コンパイル向けのリンカーオプションについて説明します。

混在モードのプロファイル

Python* はインタープリター型言語で、コンパイラーを使用しないでバイナリー実行コードを生成します。Cython も (C 拡張ではありますが) インタープリター型言語で、ネイティブコードをビルドできます。インテル® VTune™ Amplifier XE 2017 は、ホットな関数のレポートにおいて Python* コードと Cython コードの両方をサポートしています。

Cython を利用するには、次の 2 つのステップを実行します。

1. *.pyx (Cython) ファイルを *.c ファイルに変換します。
2. 変換済みの *.c ファイルを *.pyd ファイル (Linux* の .so ファイルに相当) にコンパイルします。

この例で使用するサンプルは、512 × 512 行列乗算です。

以下は、*.pyd ファイルを生成し、Windows* 環境でインテル® VTune™ Amplifier XE を使用してシンボルを解決するためのステップです。

1. Cython を使用して *.pyx ソースコードを Python* から C へ変換するため、setup.py ファイルで拡張モジュール、コンパイラー・オプション、リンクオプションを設定します。

```
#setup.py
from distutils.core import setup
from Cython.Build import cythonize
from distutils.extension import Extension
from Cython.Distutils import build_ext

setup(
      cmdclass = {'build_ext': build_ext},
      ext_modules = [Extension('Matrix_mul', sources=["Matrix_mul.pyx"],
                        extra_compile_args=['/Z7'],
                        extra_link_args=['/DEBUG']),
                  ],
      )
```

上記の setup.py ファイルでは、デバッグシンボルを生成するため、/Z7 コンパイラー・オプションと /DEBUG リンカーオプションを指定して拡張モジュールを作成しています。インテル® VTune™ Amplifier XE でソースへドリルダウンする場合、オプションを指定することが重要です。Linux* プラットフォーム上では、extra_compile_args と extra_link_args の両方で -g を指定します。

2. 次のコマンドで Python* を呼び出し、Cython を使用して C に変換します。

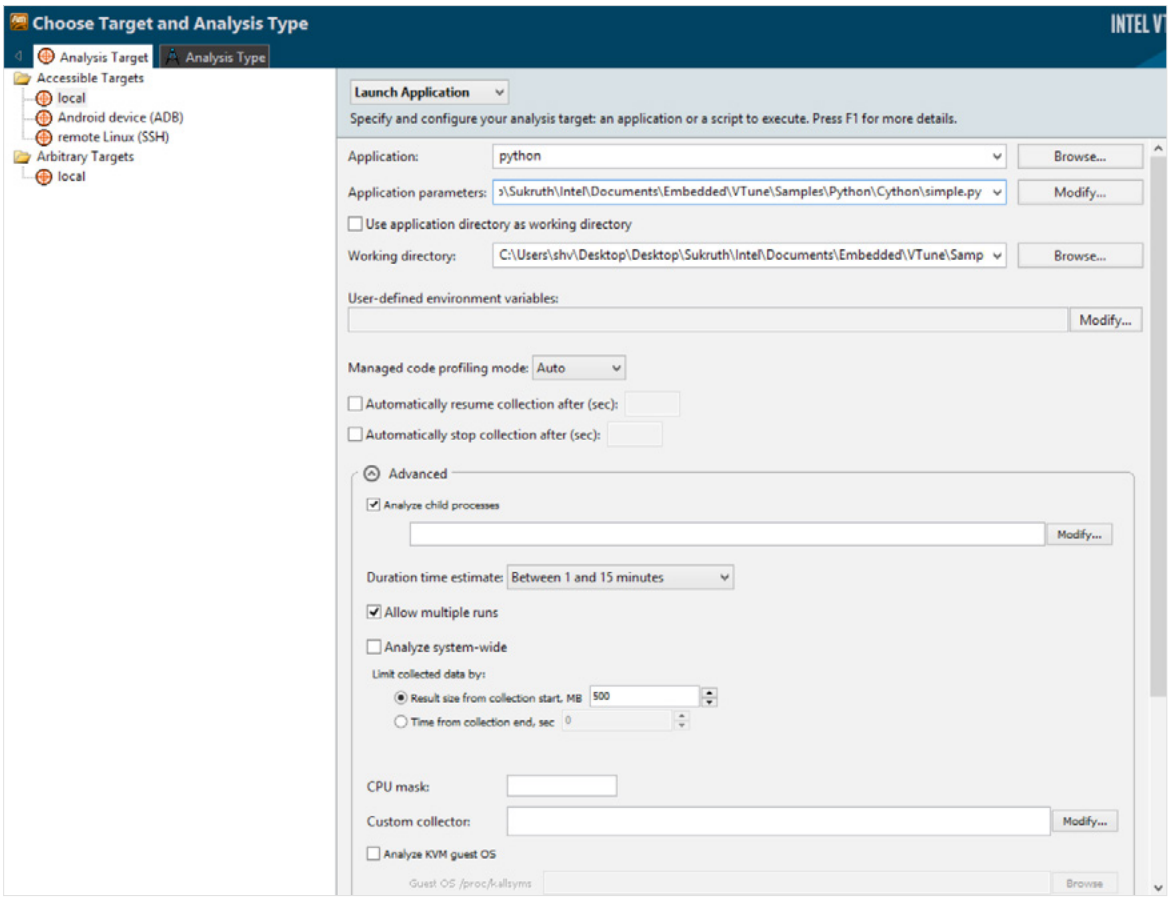
```
python setup.py build_ext
```

このコマンドは、対応する *.c と *.pyd モジュールを生成します。この例では、Matrix_mul.c と Matrix_mul.pyd が生成されます。ここでは、シンボル情報の生成も指定したため、*.pdb ファイルも生成されます。

3. Matrix_mul.pyd モジュールをインポートする Python* スクリプトを作成します。

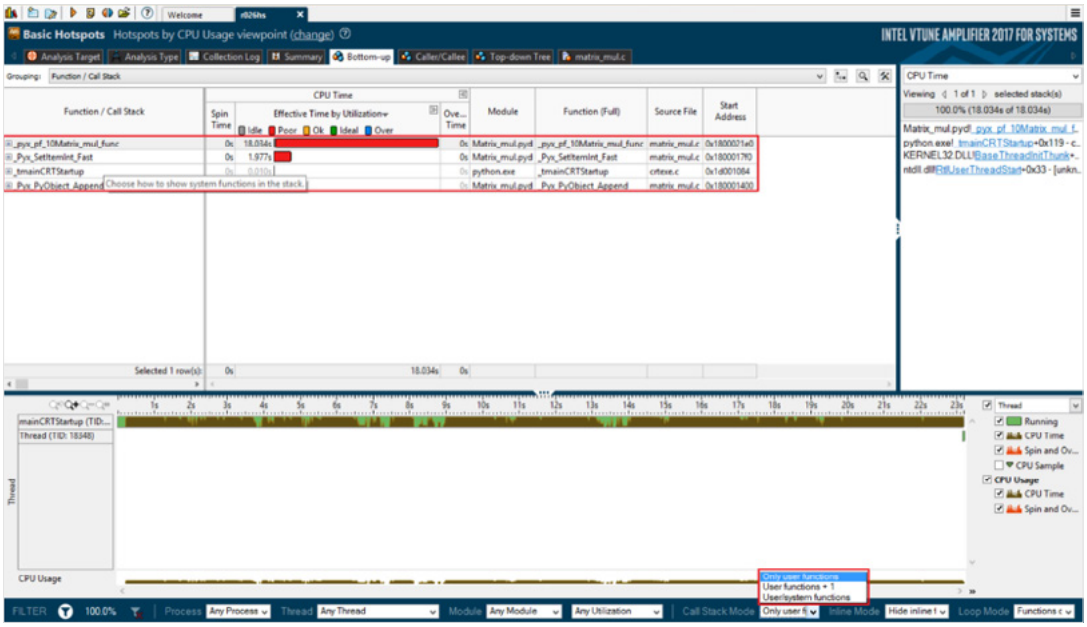
```
#simple.py
import Matrix_mul
Matrix_mul.func();
```

4. インテル® VTune™ Amplifier XE 2017 の Basic Hotspots 解析タイプで、Python* インタープリターへのパラメーターに simple.py を指定して、アプリケーションを解析します (**図 8**)。



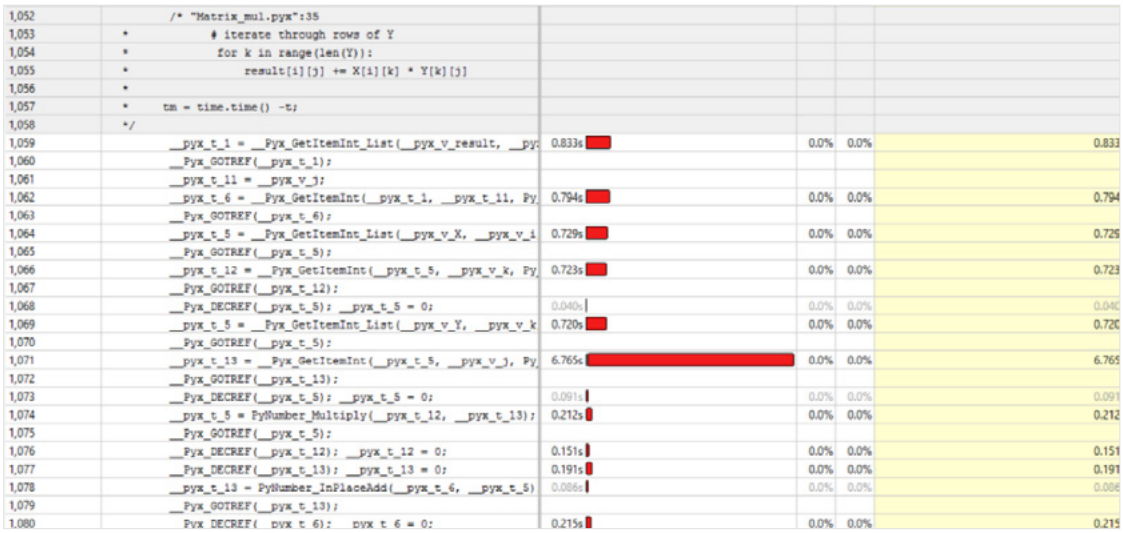
8 ターゲットと解析タイプの選択

5. Basic Hotspots 解析が完了したら、[Bottom-up] タブでモジュール / 関数を確認します (**図 9**)。



9 モジュール/関数

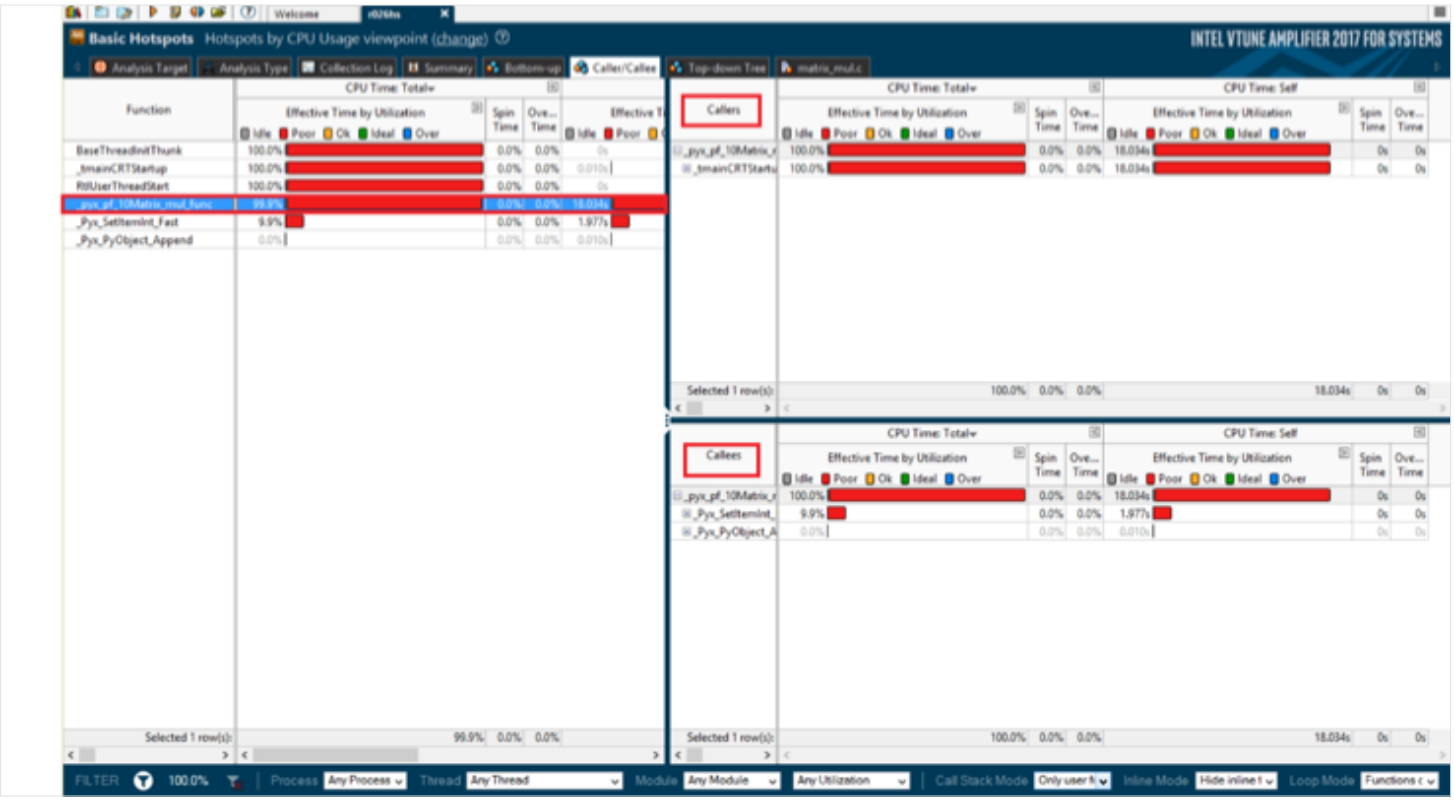
この例では、Matrix_mul.pyd モジュール (変換後のモジュール) が、多くの CPU 時間を費やしている hotspot の 1 つであることが分かります。ここでは、ユーザーコードの解析 / プロファイルのみ確認したいため、ページ下部の [FILTER] セクションにある [Call Stack Mode] で "Only user functions" (ユーザー関数のみ) を選択しています。ユーザーおよびシステムレベルの関数とそれらの相互作用を表示するには、"User/system functions" (ユーザー / システム関数) を選択します。さらに、ソースファイルにドリルダウンして、モジュールの特定のソース行を調査できます。また、C コードに対応するアセンブリーも確認できます。上記のスクリーンショットにある最初のユーザー関数 hotspot をダブルクリックすると、**図 10** のように、対応するソースファイルが表示されます。



10 最初のユーザー関数 hotspot をクリック

[Source] ビューを利用して、生成された C コードと対応する Python* コード (行 1052-1058) を関連付けることができます。このデータを使用して、インテル® VTune™ Amplifier XE で特定された Python* コードの問題の個所 / モジュールに絞って、最適化に取り掛かることができます。

[Caller/Callee] タブで、選択した関数 / モジュールの呼び出し元 - 呼び出し先を確認できます (**図 11**)。



11 [Caller/Callee] タブ

まとめ

インテル® VTune™ Amplifier XE は、Java* や Python* アプリケーションのパフォーマンス解析を、多くの CPU 時間を費やしている特定のコード (hotspot) に絞り込むことができます。これは、標準の Python* アプリケーションだけでなく、Python*/Cython が混在するコードでも可能です。

インテル® VTUNE™ AMPLIFIER XE の詳細 >

電光石火の R マシンラーニング・アルゴリズム

インテル® DAAL と最新のインテル® Xeon Phi™ プロセッサーにより成果を上げる

Zhang Zhang インテル コーポレーション ソフトウェア・テクニカル・コンサルティング・エンジニア

データ・サイエンティストは、最新のインテル® Xeon Phi™ プロセッサー x200 ファミリー (開発コード名 Knights Landing) 上で、R アプリケーションの大幅なスピードアップを実現できるようになりました。インテル® Data Analytics Acceleration Library (インテル® DAAL) のようなインテル® ソフトウェア・ツールを使用することで、開発者の労力を最小限に抑えつつ、これを達成することが可能です。

最新のインテル® Xeon Phi™ プロセッサーは、コンピューティング負荷の高いアプリケーション向けに特化したプラットフォームです。以下を含む、いくつかの新しい画期的な技術を提供します。

- ソケット (セルフブート CPU) とコプロセッサー・バージョン
- DDR4 メモリーに加えて、マルチチャネル DRAM (MCDRAM) と呼ばれる高帯域幅オンパッケージ・メモリー
- 最先端の**ベクトル化**テクノロジー : 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) 命令セット
- ダイごとに最大 72 コア、コアごとに 4 スレッドによる超並列処理
- 倍精度浮動小数点演算では 3+ TFLOPS
- 単精度浮動小数点演算では 6+ TFLOPS

インテルでは、開発者がこれらの技術を搭載したインテル® Xeon Phi™ プロセッサーを最大限に利用できるように、インテル® Parallel Studio XE スイートでソフトウェア・ツール群を提供しています。その 1 つが、最適化されたマシンラーニングおよびデータ解析アルゴリズムを提供するインテル® DAAL です。マシンラーニングの問題は、一般に CPU およびメモリー要件が最も厳しく、インテル® Xeon Phi™ プロセッサーは、このような問題にとって理想的なプラットフォームです。インテル® DAAL は、インテル® Xeon Phi™ プロセッサー向けに最適化されたマシンラーニング・アプリケーションを素早く開発できるように支援します。

R はオープンソース・プロジェクトで、統計解析、データマイニング、マシンラーニング向けの最も豊富なアルゴリズム群を提供するソフトウェア環境です。[1] R は、データ・サイエンティストの間で非常に人気があります。しかし、特別なチューニングなしでは、R とそのパッケージは、インテル® Xeon Phi ™ プロセッサーのハイパフォーマンスな機能を十分に活かすことはできません。インテル ® Xeon Phi ™ プロセッサー上での R のアウトオブボックス・パフォーマンスは良くないことが想定されます。これは、通常の R はほとんどの場合、1 つのインテル® Xeon Phi™ プロセッサー・コア上で、シングルスレッドしか使用しないためです。つまり、(72 コア、コアごとに 4 スレッドと仮定した場合) インテル® Xeon Phi™ プロセッサー上で利用可能な計算リソースの 1/288 しか使用していないことになります。1 つのインテル® Xeon Phi™ プロセッサーは、およそ 1 つのインテル® Atom™ プロセッサーに相当し、1 つのインテル® Xeon® プロセッサーよりも性能が劣ります。また、通常の R は、インテル® AVX-512 命令セットを利用する高度な**ベクトル化**や高帯域 MCDRAM のようなパフォーマンス機能を利用しません。

インテル® Xeon Phi™ プロセッサーの魅力的なデータ処理能力を、R プログラマーが簡単に利用できる方法があります。それは、R 環境に最適化されたライブラリーを統合することです。ここでは、典型的なマシンラーニング・アルゴリズムのナイーブベイズ分類器を使用します。インテル® Xeon Phi™ プロセッサー・ベースのセルフブートシステムで、インテル® DAAL を使用して、R でナイーブベイズ分類器を作成し、実行する方法をステップ・バイ・ステップで示します。そして、そのパフォーマンスを R パッケージ e1071 のネイティブ実装と比較します。[2]

ナイーブベイズ分類器

ナイーブベイズ・アルゴリズムは、ベイズの定理に基づく分類手法で、[3] すべての特徴は互いに独立していると仮定します。単純ですが、多くの場合、洗練された分類手法よりもパフォーマンスが優れています。一般に、ドキュメントの分類、スパムメールの検出などに使用されます。

特徴ベクトル $X_i=(x_{i1},\cdots,x_{ip})$、$i=1,\cdots,n$ (x_{ik} は i 番目の観測の k 番目に観測された特徴のスケーリングされた頻度、p は特徴の数)、ラベルのセット $C=(C_1,\cdots,C_d)$ の場合、ナイーブベイズ・アルゴリズムは次のようになります。

$$p(C_k|x_1,\ldots,X_n)!p(C_k)\prod_{i=1}^{n}p(x_i|C_k)$$

ラベル C_{k1} では、

事後確率は、(**事前確率 × 尤度**) に正比例します。

トレーニング段階では、各観測のラベルが判明しているトレーニング・データセットが、各クラス (ラベル) の事前確率とすべての特徴の尤度などの引数を含むモデルの学習に使用されます。そして、予測段階では、新しい観測ごとに、アルゴリズムは観測の最大事後確率を計算し、対応するラベルを割り当てます。

R のナイーブベイズ

良く知られている R のナイーブベイズ分類器は、パッケージ e1071: Misc Functions of the Department of Statistics, Probability Theory Group で提供されています。[2]

簡単なインターフェイスと 2 つの関数が含まれており、1 つはモデルをトレーニングし、もう 1 つはモデルを適用します。

```
library(e1071)

model <- naiveBayes(training_data, ground_truths)
result <- predict(model, new_data)
```

これをインテル® Xeon Phi™ プロセッサー上で実行すると、中規模のデータセット (10 万観測、200 特徴、100 クラス) のモデル・トレーニングに 200 秒以上、予測段階には 6,272 秒 (約 1 時間 45 分) かかります。

このため、パッケージ e1071 のナイーブベイズは、インテル ® Xeon Phi ™ プロセッサー上では実用的ではありません。インテル® DAAL を使用して、この問題に簡単に対処できます。

インテル® DAAL のナイーブベイズ

インテル® DAAL は、分類アルゴリズムの 1 つとして、多項ナイーブベイズ分類器を提供しています。[4] 優れたパフォーマンスに加えて、インテル® DAAL の実装は、パッケージ e1071 にはないいくつかの機能と柔軟性を備えています。特に、モデル・トレーニングでバッチ処理、オンライン処理、分散処理の 3 つの処理モードをサポートしています。バッチ処理モードは、パッケージ e1071 がサポートする処理モードと同じで、データセット全体を一度に処理します。オンライン処理モードは、一度にメモリーに収まらない大きなデータセットをチャンクごとに処理し、すべてのチャンクの処理が完了したらモデルを学習します。分散処理モードは、クラスター上での分散モデル・トレーニングをサポートします。説明を簡潔にするため、ここではバッチ処理モードのみ使用します。しかし、ここで紹介する手法は、ほかの 2 つの処理モードを R に統合する際にも適用できます。

次のセクションで説明する統合ステップを完了すると、以下のようなインテル® DAAL のナイーブベイズ分類器を使用できるようになります。

```
# インテル® DAAL のナイーブベイズ
test <- loadData("traindata.csv", nfeatures)
model.daal <- nbTrain(test$data, test$labels, nclasses)
labels.daal <- nbPredict(model.daal, test$data, nclasses)
```

インテル® DAAL のナイーブベイズ分類器の R への統合

インテル® DAAL では、(Java* と Python* API に加えて) C++ プログラミング・インターフェイスが用意されています。インテル® DAAL のナイーブベイズ・アルゴリズムを R で使用するには、トレーニングと予測段階を C++ 関数でラップし、R にエクスポートする必要があります。これには、Rcpp パッケージが役立ちます。[5, 6]

ツールのセットアップ : Rcpp と関連パッケージ

Rcpp は、C++ コードを利用して R を拡張する標準的なパッケージです。多数の R パッケージが、このパッケージを利用して C++ 実装により計算を高速化し、ほかの C++ プロジェクトと連携しています。Rcpp は R と C++ をシームレスに統合し、R と C++ の間で R オブジェクトを直接やり取りできるようにします。

Rcpp は CRAN からインストールできます。inline パッケージもインストールする必要があります。[7] inline パッケージは、R コードから直接 C++ コードをコンパイル、リンク、ロードできるようにします。Rcpp では、R で使用する C++ 関数をコンパイル、リンク、ロードする複数の方法をサポートしています。その中で使いやすさと柔軟性のバランスが良いのは、Rcpp と inline パッケージの組み合わせです。これらのパッケージに加えて、C++ **コンパイラー**も必要です。通常、インテル® Xeon Phi™ プロセッサー上でのコードのビルドおよび最適化には、インテル® C++ コンパイラーが推奨されます。しかし、ここでは多くの C++ ソースコードをビルドするわけではありません。(インテル® DAAL の) 事前ビルドバイナリーを、R でロードされる小さなダイナミック・ライブラリーにリンクするだけなので、どのコンパイラーを使用してもかまいません。システムにデフォルトの C++ コンパイラーがない場合は、次の操作を行います。

- Windows®: **Rtools** をインストールします。
- Mac OS*: App Store* から Xcode* をインストールします。
- Linux*: 次のコマンドを実行します。
  sudo apt-get install r-base-dev

R と C++ 関数を連携させるためのグルーコード

inline パッケージは、C++ 関数のシグネチャー、C++ 関数の定義、プラグイン・オブジェクトを受け取る単純な関数 cxxfunction を提供します。プラグイン・オブジェクトは、プロジェクトに必要な追加の C++ ヘッダーファイルと追加のリンク行を指定します。**図 1** に例を示します。

図 1 の行 5 - 9 は、Rcpp プラグインメーカーを使用して、R で Rcpp プラグインの作成と登録を行います。プラグインにより、外部ライブラリーへの依存関係を指定できます。この例では、インテル® DAAL を指定しています。残りのコードは、CSV ファイルからのデータの読み取り (loadData)、ナイーブベイズ・モデルのトレーニング (nbTrain)、新しいデータのラベル予測 (nbPredict) を行う 3 つの R 関数を作成します。cxxfunction を使用して、この 3 つの R 関数を、R コードで readCSV、train、predict として定義されている 3 つの C++ 関数と連携させます。

そして、インテル® DAAL のデータ構造とアルゴリズムを使用して、これらの C++ 関数を実装します。

```
1  library(Rcpp)
2  library(inline)
3
4  # Rcpp プラグインの作成と登録
5  plug <- Rcpp:::Rcpp.plugin.maker(
6          include.before = "#include <daal.h> ",
7          libs = paste("-L$DAALROOT/lib/ -ldaal_core -ldaal_thread ",
8                       "-ltbb -lpthread -lm", sep=""))
9  registerPlugin("daalNB", plug)
10
11  # データとラベルをロードする R 関数
12  loadData <- cxxfunction(signature(file="character", ncols="integer"),
13                          readCSV, plugin="daalNB")
14
15  # モデルをトレーニングする R 関数
16  nbTrain <- cxxfunction(signature(X="raw", y="raw", nclasses="integer"),
17                         train, plugin="daalNB")
18
19  # スコアリングを行う R 関数
20  nbPredict <- cxxfunction(signature(model="raw", X="raw", nclasses="integer"),
21                           predict, plugin="daalNB")
22
```

1 inline パッケージの C++ コードから R 関数を作成するグルーコード

CSV ファイルからのデータの読み取り

read.csv()、read.table()、scan() などの R 関数を使用して、CSV ファイルからデータを簡単に読み取ることができます。しかし、この方法では、読み取ったデータをインテル® DAAL で認識可能な表現に変換する必要があります。

インテル® DAAL は、インメモリー・データ表現に数値テーブル (C++ クラスの階層) を使用します。ここでは、インテル® DAAL のデータソース機能を利用してデータをロードし、そのデータを基に直接数値テーブルを作成します。**図 2** の行 13 – 16 は、トレーニング・データとその観測を含む CSV ファイルにアクセスするデータソースを定義します。CSV テーブルの最後の列は、観測 (ラベル) と仮定します。行 19 – 24 は、読み取ったデータとラベルを保持するインテル® DAAL の数値テーブルを作成します。行 27 は、データセット全体を数値テーブルにロードします。そして、R 空間に戻る前に、データ数値テーブルとラベル数値テーブルを 2 つの RAW バイトにシリアル化し、それらのリストを返します。後で、C++ モデル・トレーニング関数が RAW バイトを受け取り、数値テーブルを復元します。

```
# データのロード
readCSV <- '
    using namespace daal;
    using namespace daal::data_management;

    // 入力
    // file - ファイル名
    // ncols - ファイルに含まれる列数
    std::string fname = Rcpp::as<std::string>(file);
     int k = Rcpp::as<int>(ncols);

     // データソース
     FileDataSource<CSVFeatureManager> dataSource(
          fname,
          DataSource::notAllocateNumericTable,
          DataSource::doDictionaryFromContext);

     // インテル® DAAL のデータおよびラベル数値テーブル
     services::SharedPtr<NumericTable> data(
         new HomogenNumericTable<double>(k-1, 0, NumericTable::notAllocate));
     services::SharedPtr<NumericTable> labels(
         new HomogenNumericTable<int>(1, 0, NumericTable::notAllocate));
     services::SharedPtr<NumericTable> merged(
         new MergedNumericTable(data, labels));

     // データのロード
     dataSource.loadDataBlock(merged.get());

     // 数値テーブルのシリアル化
     InputDataArchive dataArch, labelsArch;
     data->serialize(dataArch);
     labels->serialize(labelsArch);
     Rcpp::RawVector dataBytes(dataArch.getSizeOfArchive());
     dataArch.copyArchiveToArray(&dataBytes[0], dataArch.getSizeOfArchive());
     Rcpp::RawVector labelsBytes(labelsArch.getSizeOfArchive());
     labelsArch.copyArchiveToArray(&labelsBytes[0], labelsArch.getSizeOfArchive());

     // RawVectors のリストを返す
     return Rcpp::List::create(
                               _["data"]   = dataBytes,
                               _["labels"] = labelsBytes);
 '
```

2 CSV ファイルからデータを読み取り、インテル® DAAL の数値テーブルを作成する C++ 関数

ナイーブベイズ・モデルのトレーニング

インテル® DAAL のアルゴリズムをコードで使用する場合、次の簡単なステップに従って、一連の処理を実装できます。

1. 選択したアルゴリズムと処理モード向けのアルゴリズム・オブジェクトを作成します。
2. アルゴリズム・オブジェクトの input.set メソッドを使用して、入力データを設定します。
3. アルゴリズム・オブジェクトで compute メソッドを呼び出します。
4. アルゴリズム・オブジェクトの getResult メソッドを使用して、結果を取得します。

図 3 の行 25 – 28 は、C++ の train 関数の実装におけるこの一連の処理を示します。これ以前のコードは、最初に入力 (トレーニング・データとラベル) を逆シリアル化し、数値テーブルを復元します。これ以降のコードは、結果 (モデル・オブジェクト) を RAW バイトにシリアル化します。そうすることで、新しいデータの分類に使用されるモデルを予測関数に渡すことができます。

```
1  # ナイーブベイズ: モデルのトレーニング
2  train <- '
3      using namespace daal;
4      using namespace daal::algorithms;
5      using namespace daal::algorithms::multinomial_naive_bayes;
6      using namespace daal::data_management;
7
8      // 入力
9      // x - トレーニング・データセット
10      // y - トレーニング・データの観測
11      // nclasses - クラスの数
12      Rcpp::RawVector Xr(X);
13      Rcpp::RawVector yr(y);
14      int nClasses = Rcpp::as<int>(nclasses);
15
16      // データとラベルの逆シリアル化
17      OutputDataArchive dataArch(&Xr[0], Xr.length());
18      services::SharedPtr<NumericTable> ntData(new HomogenNumericTable<double>());
19      ntData->deserialize(dataArch);
20      OutputDataArchive labelsArch(&yr[0], yr.length());
21      services::SharedPtr<NumericTable> ntLabels(new HomogenNumericTable<int>());
22      ntLabels->deserialize(labelsArch);
23
24      // モデルのトレーニング
25      training::Batch<> algorithm(nClasses);
26      algorithm.input.set(classifier::training::data, ntData);
27      algorithm.input.set(classifier::training::labels, ntLabels);
28      algorithm.compute();
29
30      // 結果の取得
31      services::SharedPtr<training::Result> result = algorithm.getResult();
32      InputDataArchive archive;
33      result->get(classifier::training::model)->serialize(archive);
34
35      Rcpp::RawVector out(archive.getSizeOfArchive());
36      archive.copyArchiveToArray(&out[0], archive.getSizeOfArchive());
37      return out;
38  '
```

3 ナイーブベイズ・モデルをトレーニングする C++ 関数

新しいデータのラベル予測

図 4 は、C++ の predict 関数です。行 28 – 31 に、train 関数と同様のコードシーケンスがあります。予測の結果は、予測されたラベルを保持する n×1 数値テーブルです。ここで、n は予測の観測数です。図 4 の最後のコードブロックは、結果の数値テーブルからラベルを読み取り、Rcpp::IntegerVector オブジェクトに格納します。このオブジェクトは、R 空間へシームレスに渡され、整数配列になります。

```
1  # ナイーブベイズ: 予測
2  predict <- '
3      using namespace daal;
4      using namespace daal::algorithms;
5      using namespace daal::algorithms::multinomial_naive_bayes;
6      using namespace daal::data_management;
7
8      // 入力
9      // model - トレーニング済みモデル
10      // x - 入力データ
11      // nclasses - クラスの数
12      Rcpp::RawVector modelBytes(model);
13      Rcpp::RawVector dataBytes(X);
14      int nClasses = Rcpp::as<int>(nclasses);
15
16      // モデルの取得
17      OutputDataArchive modelArch(&modelBytes[0], modelBytes.length());
18      services::SharedPtr<multinomial_naive_bayes::Model> nb(
19          new multinomial_naive_bayes::Model());
20      nb->deserialize(modelArch);
21
22      // データの逆シリアル化
23      OutputDataArchive dataArch(&dataBytes[0], dataBytes.length());
24      services::SharedPtr<NumericTable> ntData(new HomogenNumericTable<double>());
25      ntData->deserialize(dataArch);
26
27      // 新しいデータの予測
28      prediction::Batch<> algorithm(nClasses);
29      algorithm.input.set(classifier::prediction::data,  ntData);
30      algorithm.input.set(classifier::prediction::model, nb);
31      algorithm.compute();
32
33      // newlabels を返す
34      services::SharedPtr<NumericTable> predictionResult =
35          algorithm.getResult()->get(classifier::prediction::prediction);
36      BlockDescriptor<int> block;
37      int n = predictionResult->getNumberOfRows();
38      predictionResult->getBlockOfRows(0, n, readOnly, block);
39      int* newlabels = block.getBlockPtr();
40      IntegerVector predictedLabels(n);
41      std::copy(newlabels, newlabels+n, predictedLabels.begin());
42      return predictedLabels;
43  '
```

4 トレーニング済みモデルを使用して新しいデータのラベルを予測する C++ 関数

すべてのコードの統合

図 1 – 4 のコードを NaiveBayesClassifierDaal.R のように 1 つのスクリプトにまとめることができます。そして、R の source() 関数を使用して、このスクリプトを R 環境で利用できます。前述の 3 つの関数に対応する新しい関数は loadData()、nbTrain()、nbPredict() です。**図 5** は、R でこれらの関数を使用する方法を示します。また、コード例では、トレーニングと予測段階のパフォーマンス・ベンチマークを測定するため、microbenchmark() を使用しています。

R でスクリプトを source するたびに、C++ コードを R 拡張にビルドするため、自動的にコンパイルとリンクが行われます。これには時間がかかります。このオーバーヘッドを回避するには、Rcpp を使用して専用の R パッケージを記述することを検討すべきです。

専用の R パッケージには、R へのロード時にコンパイルとリンクが不要な事前にビルドされたダイナミック・ライブラリーを含めます。ここで示したインテル® DAAL を使用するための C++ コードを R パッケージのビルドに利用することができます。R パッケージの記述方法については、記事の趣旨から外れるため、ここでは説明しません。興味がある方は、公式の『Writing R Extensions』マニュアルで詳細を確認してください。[8]

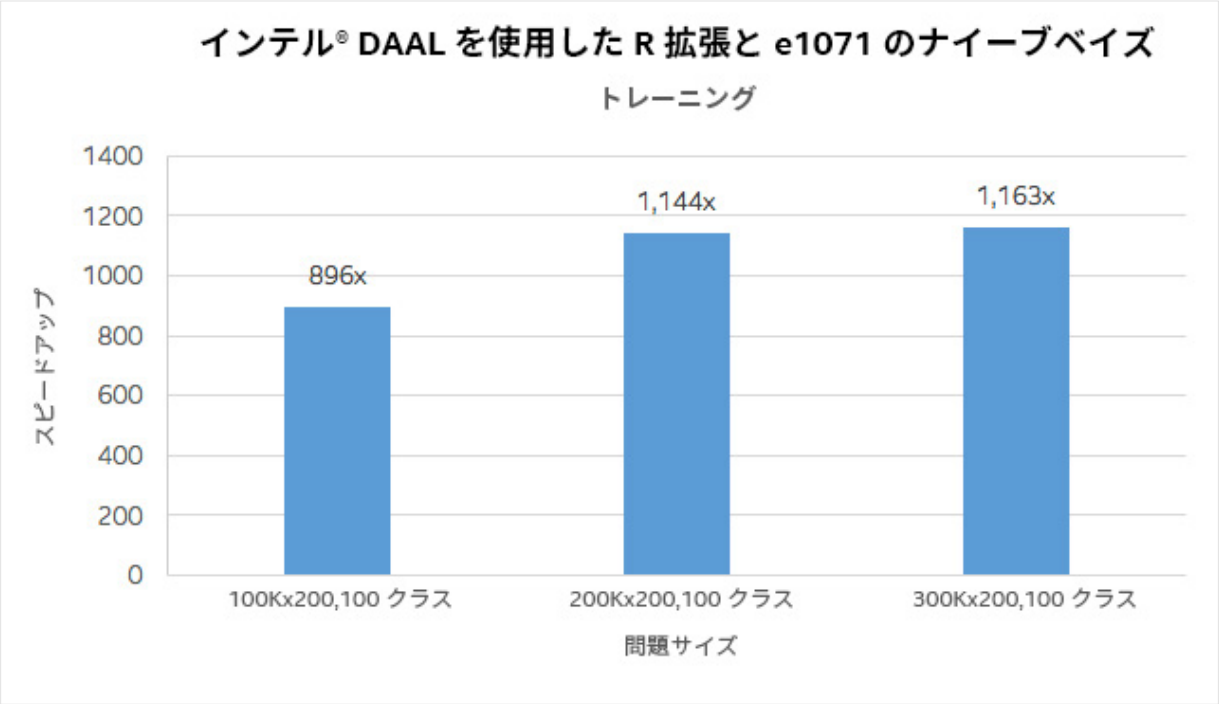
```
1  source("NaiveBayesClassifierDaal.R")
2
3  # インテル® DAAL のナイーブベイズ
4  test <- loadData("traindata.csv", nfeatures)
5  trainperf.daal <- microbenchmark(
6              model.daal <- nbTrain(test$data, test$labels, nclasses))
7  scoreperf.daal <- microbenchmark(
8              labels.daal <- nbPredict(model.daal, test$data, nclasses))
```

5 インテル® DAAL を利用して作成されたナイーブベイズ分類器の使用

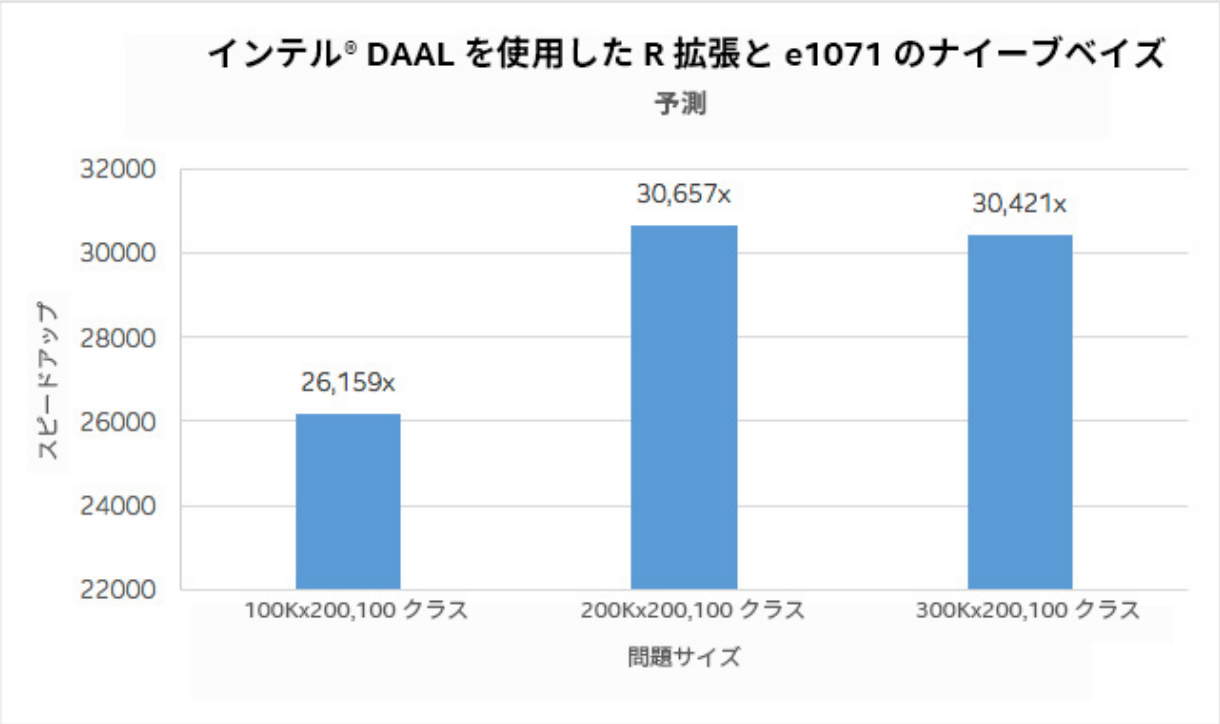
パフォーマンス・ゲイン

ここで紹介したソリューションは、モデル・トレーニングと予測のパフォーマンスを大幅に向上します。前述のように、パッケージ e1071 のナイーブベイズは、トレーニングに 200 秒以上、予測に 1 時間 45 分かかります。**一方、ここで紹介した実装は、2 つの段階を 0.25 秒以内で完了できます。**

インテル® DAAL のナイーブベイズを使用する R 拡張実装と、パッケージ e1071 のナイーブベイズを、いくつかのデータセットで実行してみたところ、インテル® DAAL を使用する実装は、トレーニング段階では 1,000 倍、予測段階では最大 30,000 倍のスピードアップをもたらしました。**グラフ 1** と**グラフ 2** に各データセットでのスピードアップを示します。



グラフ 1. パッケージ e1071 のナイーブベイズ分類器と比較したスピードアップ (トレーニング段階)



グラフ 2. パッケージ e1071 のナイーブベイズ分類器と比較したスピードアップ (予測段階)

まとめ

インテル® Xeon Phi™ プロセッサーは、マシンラーニングとデータ解析ワークロードに理想的なハイパフォーマンスな超並列プラットフォームです。しかし、データサイエンス・コミュニティーで人気がある R のような主要ソフトウェア・ツールは、またインテル® Xeon Phi™ プロセッサーで効率良く実行できるようにチューニングされていません。ここでは、インテル® Xeon Phi™ プロセッサーを今すぐ利用したいと考えている R プログラマー向けにソリューションを紹介しました。

Rcpp を使用して R 拡張を記述することで、インテル® DAAL 関数を R 環境に統合することができます。インテル® DAAL は、データ解析、マシンラーニング、ディープラーニング向けのすぐに使えるアルゴリズムを提供するライブラリーです。これらのアルゴリズムは、インテル® Xeon Phi™ プロセッサーだけでなく、インテル® Xeon® プロセッサー、インテル® Core™ プロセッサー、インテル® Atom™ プロセッサー向けに最適化されています。ベンチマーク結果は、インテル® DAAL を使用するソリューションのほうが、e1071 パッケージのネイティブ R ソリューションよりも優れたパフォーマンスを達成できることを示しています。ここで紹介した手法により、多くの R アプリケーションでインテル® Xeon Phi™ プロセッサーの利点をすぐに利用することができます。

ここでは、例として多項ナイーブベイズ・アルゴリズムを使用しましたが、この手法を利用して、ほかのインテル® DAAL アルゴリズムを R に統合することができます。インテル® DAAL は、線形回帰、SVM、ナイーブベイズ、ブースティングを利用する分類、推奨システム、クラスタリング、ディープ・ニューラル・ネットワークを含む、豊富なアルゴリズムのセットを提供します。

Github* のインテル® DAAL オープンソース・プロジェクト (英語) >

システム構成と使用したツール

ベンチマークのシステム構成 :

- インテル® Xeon Phi™ プロセッサー (セルフブート) ベースのシステム
  - CPU: インテル® Xeon Phi™ プロセッサー D B0、68 コア @ 1.40GHz、34MB L2 キャッシュ
  - メモリー : 16GB MCDRAM、96GB DDR4
- OS バージョン : Red Hat* Enterprise Linux* 7.2 (カーネル 3.10.0-327.0.1.el7.x86_64、glibc 2.17-105.el7.x86_64)

例で使用したソフトウェア・ツール :

- インテル® DAAL 2017 Beta Update 1
- R 3.3.1 (Bug in Your Hair)
- Rcpp パッケージ 0.12.5
- inline パッケージ 0.3.14
- e1071 パッケージ 1.6-7
- g++ 4.8.5

参考資料 (英語)

1. The R* Project for Statistical Computing, **r-project.org/**.
2. e1071: Misc Functions of the Department of Statistics, Probability Theory Group, David Meyer, **cran.r-project.org/web/packages/e1071/index.html**.
3. Electronic Statistics Textbook, StatSoft, Inc., Tulsa, OK, 2013.
4. Developer Guide and Reference for Intel® Data Analytics Acceleration Library 2016, **software.intel.com/sites/products/documentation/doclib/daal/daal-user-and-reference-guides/index.htm**.
5. Rcpp: Seamless R* and C++ Integration, Dirk Eddelbuettel, **cran.r-project.org/web/packages/Rcpp/index.html**.
6. Seamless R* and C++ Integration with Rcpp, D. Eddelbuettel, Springer, 2013.
7. inline: Functions to Inline C, C++, Fortran Function Calls from R*, Oleg Sklyar, **cran.r-project.org/web/packages/inline/index.html**.
8. Writing R* Extensions, **cran.r-project.org/doc/manuals/r-release/R-exts.html**.

データ解析およびマシンラーニング向けパフォーマンス・ライブラリー

インテル® DAAL による手書き数字認識向け C++ コードの作成例

Shaojuan Zhu インテル コーポレーション ソフトウェア・テクニカル・コンサルティング・エンジニア

インテル® Data Analytics Acceleration Library (インテル® DAAL) は、インテル® プラットフォーム上でのデータ解析とマシンラーニング向けに最適化されたビルディング・ブロックを提供する、データ解析用のパフォーマンス・ライブラリーです。インテル® DAAL に含まれる関数は、マシンラーニングのすべてのデータ処理段階 (前処理、変換、解析、モデリングから意思決定まで) をカバーします。各処理段階向けに、インテル® Atom™ プロセッサー、インテル® Core™ プロセッサー、インテル® Xeon® プロセッサー、インテル® Xeon Phi™ プロセッサーを含むインテル® アーキテクチャー向けに最適化された関数とユーティリティーが用意されています。インテル® DAAL は、3 つの処理モード (バッチ、オンライン、分散) と 3 つのプログラミング API (C++、Java*、Python*) をサポートします。

ここでは、インテル® DAAL の手書き数字認識アプリケーションの C++ コード例について見ていきます。手書き数字認識は、典型的なマシンラーニングの問題の 1 つで、いくつかのアプリケーション・アルゴリズムに関連があります。サポート・ベクトル・マシン (SVM)、主成分分析 (PCA)、ナイーブベイズ、ニューラル・ネットワークはすべて、この問題への対応に使用されますが、予測の精度が異なります。インテル® DAAL には、これらのアルゴリズムのほとんどが含まれています。ここでは、SVM を使用して、手書き数字認識にインテル® DAAL のアルゴリズムを導入する方法を示します。

インテル® DAAL でのデータのロード

手書き数字認識において、認識は基本的にマシンラーニング・パイプラインの予測 / 推論段階に当たります。提供された手書き数字から、システムが数字を認識または推論できなければなりません。システムが入力から出力を予測 / 推論できるように、トレーニング・データセットから学習したトレーニング済みモデルが必要になります。トレーニング・モデルを作成する前のステップとして、最初にトレーニング・データを収集します。

サンプル・アプリケーションでは、**UCI Machine Learning Repository** (英語) で公開されている 3,823 件の前処理済みトレーニング・データと 1,797 件の前処理済みテストデータを使用します。インテル® DAAL は、いくつかのデータ形式 (CSV、SQL、HDFS、KDB+) とユーザー定義のデータ形式をサポートしています。ここでは、CSV 形式を使用します。トレーニング・データは `digits_tra.csv` ファイルに、テストデータは `digits_tes.csv` に保存されると仮定します。

インテル® DAAL には、データソースからメモリーへデータをロードするためのいくつかのユーティリティーがあります。ここでは、最初に `trainDataSource` オブジェクトを定義します。これは、CSV ファイルからメモリーへデータをロードすることができる `CSVFeatureManager` です。インテル® DAAL の内部では、メモリーにあるデータは数値テーブルとして保持されます。`CSVFeatureManager` を利用すると、自動的に数値テーブルが作成されます。CSV ファイルからデータをロードするには、メンバー関数 `loadDataBlock()` を呼び出します。これで、データが数値テーブルとしてメモリーにロードされ、後続の処理を行うことができます。CSV ファイルからトレーニング・データをロードする C++ コードの重要な部分を**図 1** に示します。

```
/* 入力データセットのパラメーター */
     string trainDatasetFileName = "digits_tra.csv";
     string testDatasetFileName = "digits_tes.csv";

/* FileDataSource<CSVFeatureManager> を初期化して .csv ファイルから入力データを取得 */
     FileDataSource<CSVFeatureManager> trainDataSource(trainDatasetFileName,
     DataSource::doAllocateNumericTable, DataSource::doDictionaryFromContext);

/* データファイルからデータをロード */
     trainDataSource.loadDataBlock(nTrainObservations);
```

1 トレーニング・データをロードする C++ コード

SVM ベースの手書き数字認識モデルのトレーニング

トレーニング・データがメモリーにロードされたら、そのデータを学習して、トレーニング・モデルを作成します。ここでは、トレーニング・データセットが小さく、データを一度にメモリーに収めることができるため、トレーニング用のアルゴリズムとして、SVM を使用します。処理には、バッチ処理モードを使用します。アルゴリズムを定義後、クラスの数 (この例では 10) などのアルゴリズムに必要な関連パラメーターを設定します。そして、トレーニング・データの数値テーブル `trainDataSource.getNumericTable()` をアルゴリズムに渡します。

`algorithm.compute()` を呼び出すと、SVM の計算が開始され、しばらくするとトレーニングが完了します。トレーニング済みモデルは、`trainingResult` オブジェクトに格納され、`algorithm.getResult()` を呼び出して取得できます。**図 2** にトレーニング・プロセスのサンプルコードを示します。

```
services::SharedPtr<svm::training::Batch<> > training(new
  svm::training::Batch<>());

/* 多クラス SVM トレーニング用のアルゴリズム・オブジェクトを作成 */
    multi_class_classifier::training::Batch<> algorithm;
    algorithm.parameter.nClasses = nClasses;
    algorithm.parameter.training = training;

/* トレーニング・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,
    trainDataSource.getNumericTable());

/* 多クラス SVM モデルを作成 */
    algorithm.compute();

/* アルゴリズムの結果を取得 */
    trainingResult = algorithm.getResult();

/* 学習したモデルをディスクファイルへシリアル化 */
    ModelFileWriter writer("./model");
    writer.serializeToFile(trainingResult->get(classifier::training::model));
```

2 トレーニング・プロセスのサンプルコード

インテル® DAAL は、トレーニング済みモデルをメモリーからファイルへ出力するシリアル化関数と、トレーニング済みモデルファイルをメモリーにロードする逆シリアル化関数を提供します。**図 2** の最後の 2 行のように、`model` という名前のファイルに書き込む `ModelFileWriter` を定義します。`writer.serializeToFile()` を呼び出して、`trainingResult` に格納されているトレーニング済みモデルを `model` ファイルに書き込みます。このシリアル化 / 逆シリアル化ユーティリティーは、トレーニング後、サーバーがトレーニング済みモデルをクライアントへ移植し、クライアントがトレーニングを行わずにそれを使用して予測 / 推論を行う場合に役立ちます。`model` ファイルの利用法については、「手書き数字認識アプリケーション」セクションで説明します。

トレーニング済みモデルのテスト

トレーニング済みモデルを使用してテストを行うことができます。トレーニング・プロセスと同様に、**UCI Machine Learning Repository** (英語) からのテストデータを CSV ファイルに保存し、`testDataSource.loadDataBlock()` を利用してロードします。予測用の SVM アルゴリズム・オブジェクト `algorithm` を定義する必要があります。`algorithm` には、テストデータとトレーニング済みモデルの 2 つの入力があります。テストデータは、`testDataSource.getNumericTable()` で渡します。バッチテストでは、トレーニング済みモデルは `trainingResult->get()` で渡します。(トレーニング済みモデルをファイルで渡す方法は、「手書き数字認識アプリケーション」セクションで示します。)

アルゴリズムの入力の設定が完了したら、`algorithm.compute()` を呼び出してテストプロセスを完了します。**図 3** は、テストプロセスのサンプルコードです。テスト後の予測結果は、`algorithm.getResult()` を呼び出して取得します。

```
services::SharedPtr<svm::prediction::Batch<> > prediction(new
  svm::prediction::Batch<>());

/* FileDataSource<CSVFeatureManager> を初期化して .csv ファイルからテストデータを取得 */
        FileDataSource<CSVFeatureManager> testDataSource(testDatasetFileName,
                DataSource::doAllocateNumericTable,
                DataSource::doDictionaryFromContext);
        testDataSource.loadDataBlock(nTestObservations);

/* 多クラス SVM 値の予測用のアルゴリズム・オブジェクトを作成 */
        multi_class_classifier::prediction::Batch<> algorithm;

        algorithm.parameter.prediction = prediction;

/* テスト・データセットとトレーニング済みモデルをアルゴリズムに渡す */
        algorithm.input.set(classifier::prediction::data,
                testDataSource.getNumericTable());
        algorithm.input.set(classifier::prediction::model,
                trainingResult->get(classifier::training::model));

/* 多クラス SVM 値を予測 */
        algorithm.compute();

/* アルゴリズムの結果を取得 */
        predictionResult = algorithm.getResult();
```

3 テストプロセスのサンプルコード

インテル® DAAL には、混同行列、平均正解率、不正解率などの品質メトリックを計算する関数も含まれています。**図 4** は、**UCI Machine Learning Repository** (英語) からのテスト・データセットに対し SVM を利用した場合の品質メトリックです。テストデータ全体の平均正解率が 99.6% であることが分かります。

Confusion matrix:

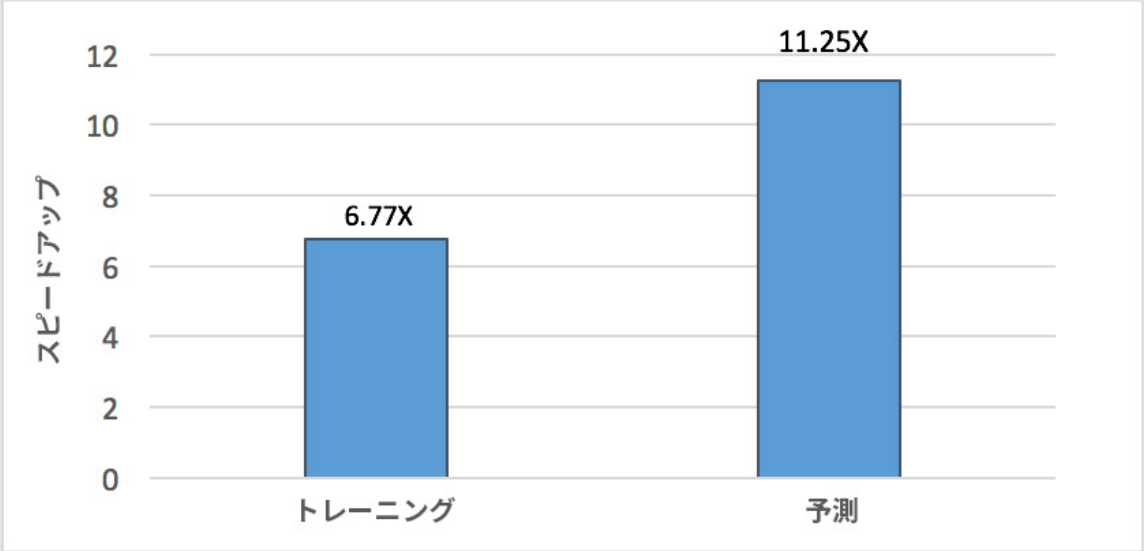
177.000	0.000	0.000	0.000	1.000	0.000	0.000	0.000	0.000	0.000
0.000	181.000	0.000	0.000	0.000	0.000	0.000	0.000	1.000	0.000
0.000	2.000	173.000	0.000	0.000	0.000	0.000	1.000	1.000	0.000
0.000	0.000	0.000	176.000	0.000	1.000	0.000	0.000	3.000	3.000
0.000	1.000	0.000	0.000	179.000	0.000	0.000	0.000	1.000	0.000
0.000	0.000	0.000	0.000	0.000	180.000	0.000	0.000	0.000	2.000
0.000	0.000	0.000	0.000	0.000	0.000	180.000	0.000	1.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	170.000	1.000	8.000
0.000	3.000	0.000	0.000	0.000	0.000	0.000	0.000	166.000	5.000
0.000	0.000	0.000	2.000	0.000	1.000	0.000	0.000	2.000	175.000

Average accuracy: 0.996
Error rate: 0.004
Micro precision: 0.978
Micro recall: 0.978
Micro F-score: 0.978
Macro precision: 0.978
Macro recall: 0.978
Macro F-score: 0.978

4 品質メトリック

インテル® DAAL と scikit-learn の SVM パフォーマンスの比較

SVM アルゴリズムは、多くのマシンラーニング・フレームワークやライブラリーのパッケージで採用されている古典的なアルゴリズムです。scikit-learn は、よく用いられるマシンラーニング向けの Python* ライブラリーです。ここでは、scikit-learn の SVM 分類を利用する手書き数字アプリケーションで同じトレーニングとトレーニング・データを使用し、インテル® DAAL と scikit-learn の SVM パフォーマンス (トレーニングと予測) を比較しました。結果は、**図 5** に示すとおりです。



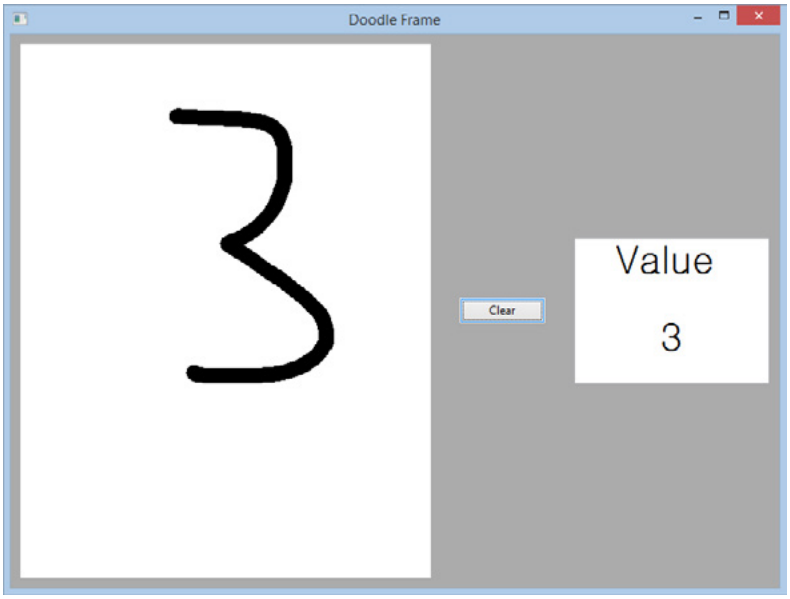
5 インテル® プロセッサー上でのインテル® DAAL と scikit-learn の SVM パフォーマンスの向上

テストに使用したシステムは、インテル® Xeon® プロセッサー E5-2680 V3 @ 2.50GHz、24 コア、CPU ごとに 30MB L3 キャッシュ、256GB RAM です。オペレーティング・システムは、Red Hat* Enterprise Linux* Server 6.6 (64 ビット) です。各ライブラリーのバージョンは、インテル® DAAL 2016 Update 2 と scikit-learn 0.16.1 です。

図 5 に示すように、トレーニングとテストの両方において、インテル® DAAL の SVM パフォーマンスのほうが scikit-learn よりも優れています。インテル® DAAL の SVM は scikit-learn よりも、トレーニングでは 6.77 倍、テスト / 予測では 11.25 倍高速です。

手書き数字認識アプリケーション

前述のように、サーバー側で学習したトレーニング済みモデルを、クライアントに移植して使用できます。ここでは、トレーニング済みモデルを移植可能な、単純な手書き数字認識アプリケーションを使用します。このインタラクティブなアプリケーションには、予測 / テスト段階しかありません。**図 6** は、アプリケーションのインターフェイスです。2 つの白いパネルボックスがあります。左の大きな白いパネルボックスに、0 から 9 の数字を 1 つ書き込みます。右の小さな白いパネルボックスは、システムが認識した左のパネルボックスに書き込まれた数字を表示します。**図 6** では、左のパネルボックスに 3 と書き込まれており、システムはこれを認識して、推論結果 3 を右のパネルボックスに表示しています。



6 数字認識アプリケーション

このアプリケーションでは、一度に 1 つの数字のみテスト / 推論します。書き込まれた数字に対し、いくつかの前処理手法により手書き数字から抽出された特徴を含むテスト CSV ファイルが生成されます。(前処理は、この記事の趣旨から外れるため、ここでは取り上げません。) テストデータを取得できたので、algorithm オブジェクトのもう 1 つの入力であるトレーニング済みモデルについて見ていきましょう。

前述のように、トレーニング済みモデルは作成済みで、model ファイルにあります。ここでは、トレーニング済みモデルをファイルからロードします (つまり、モデルをメモリーへ逆シリアル化します)。ModelFileReader を定義して、reader.deserializeFromFile(pModel) を呼び出し、model ファイルから読み取ります。pModel はモデルへのポインターです。**図 7** に C++ コードを示します。大部分のコードは、**図 3** と同じです。algorithm.compute() が完了したら、入力された数字に対するラベル / 予測された数字を含む予測結果 predictionResult1 を取得します。

```
services::SharedPtr<svm::prediction::Batch<> > prediction1(new
  svm::prediction::Batch<>());

/* FileDataSource<CSVFeatureManager> を初期化して .csv ファイルからテストデータを取得 */
        FileDataSource<CSVFeatureManager> testDataSource(testDatasetFileName,
                DataSource::doAllocateNumericTable,
                DataSource::doDictionaryFromContext);
        testDataSource.loadDataBlock(1);

/* 多クラス SVM 値の予測用のアルゴリズム・オブジェクトを作成 */
        multi_class_classifier::prediction::Batch<> algorithm;

        algorithm.parameter.prediction = prediction1;

/* ディスクファイルからモデルを逆シリアル化 */
        ModelFileReader reader("./model");
        services::SharedPtr<multi_class_classifier::Model> pModel(new
multi_class_classifier::Model());
        reader.deserializeFromFile(pModel);

/* テスト・データセットとトレーニング済みモデルをアルゴリズムに渡す */
        algorithm.input.set(classifier::prediction::data,
testDataSource.getNumericTable());
        algorithm.input.set(classifier::prediction::model, pModel);

/* 多クラス SVM 値を予測 */
        algorithm.compute();

/* アルゴリズムの結果を取得 */
        predictionResult1 = algorithm.getResult();

/* 予測されたラベルを取得 */
        predictedLabels1 = predictionResult1->get(classifier::prediction::prediction);
```

7 C++ コード

まとめ

インテル® DAAL は、マシンラーニングのパイプライン全体に対応したビルディング・ブロックを提供します。ここでは、SVM の C++ コード例を用いて、アプリケーションでインテル® DAAL を使用して、ファイルからデータをロードし、トレーニング・モデルを作成し、トレーニング済みモデルを適用する方法を紹介しました。

インテル® DAAL の関数はインテル® アーキテクチャー向けに最適化されているため、マシンラーニング・アプリケーションにインテル® DAAL のビルディング・ブロックを利用することで、インテル® プラットフォーム上で最良のパフォーマンスが得られます。ここで紹介したように、インテル® DAAL の SVM は、scikit-learn の SVM よりもパフォーマンスが優れています。

ケーススタディー

インテルのハイパフォーマンス・ライブラリーにより MERITDATA 社が TEMPO* ビッグデータ・プラットフォームをスピードアップ

Jin Qiang MeritData, Inc. データマイニング・アルゴリズム・アーキテクト
Ying Hu インテル APAC R&D Ltd. テクニカル・コンサルティング・エンジニア
Ning Wang インテル China フィールド・アプリケーション・エンジニア

MeritData, Inc. は、ビッグデータ解析技術およびサービス分野における中国の大手プロバイダーです。MeritData の Tempo* ビッグデータ・プラットフォームは、大手の電力、製造、金融、グローバル企業、クラウド・サービス・プロバイダーで広く採用されています。ハイパフォーマンス・コンピューティング技術、最先端のデータ解析アルゴリズム、高次元の視覚化、独創的なデータ可視化言語を融合することで、MeritData は顧客がデータの価値を発見•利用できるようにし、最終的にデータ処理、データマイニング、データ可視化ソリューションにより価値を創造できるように支援します。

データを素早く正確に解析するため、MeritData はアルゴリズムを基本的性質、計算、プログラミングの観点から最適化する必要がありました。インテルは MeritData のアルゴリズム・エンジニアと協力して、複数のデータマイニング・アルゴリズムの最適化に取り組みました。エクストリーム・ラーニング・マシン (ELM: Extreme Learning Machines) や内製の L1/2 スパース反復アルゴリズム、線形回帰 (LR) はすべて、インテル® Data Analytics Acceleration Library (インテル® DAAL) とインテル® マス・カーネル・ライブラリー (**インテル® MKL**) により最適化されました。その結果、パフォーマンスが平均で 3 倍～ 14 倍向上しました。

ビッグデータ解析プラットフォームで求められるスピード

ハードウェアと情報技術の開発は、ビッグデータの新しい時代を切り開きました。グローバルデータの飛躍的な成長により、ビッグデータ解析とビッグデータ・サービスの市場も拡大しています。ほぼすべての大企業がビッグデータ解析にリソースを投資しており、長年にわたって蓄積されたデータと新たに生成されるデータを統合し、素早く正確にデータの価値を引き出したいと考えています。

業界最大手の大規模データ解析技術プロバイダーとして、MeritData は顧客のあらゆるデータを格納・処理できる Tempo* ビッグデータ解析プラットフォームを提供しています。データを効率良く解析し、より多くのデータをより迅速に処理するためには、MeritData はアルゴリズムのパフォーマンスを極限まで向上する必要がありました。

「インテルのエンジニアと緊密に協力して、我々はビッグデータ解析プラットフォーム (Tempo*) のアルゴリズムの最適化にインテル® DAAL とインテル® MKL を採用しました。その結果、パフォーマンスと顧客のエクスペリエンスが大幅に向上しました。インテルの協力に大変感謝しています。今後も、インテルと協力していきたいと思います。」

MeritData
データマイニング・アルゴリズム・アーキテクト
Jin Qiang 氏

アルゴリズムのモデル化は、入力データに対し繰り返し計算を行う計算負荷の高い処理を支援します。データ量が少ない場合、通常実行時間は問題になりません。しかし、データ量の増加に伴って、一部のアルゴリズムは実行時間が急激に増加し、顧客の要件を満たすことができなくなります。

高まるデータマイニングへの要求に応えるため、MeritData はインテルと緊密に協力し、**インテル® MKL** とインテル® DAAL を利用して、Tempo* のコア・アルゴリズム・ライブラリーを高速化し、顧客に強力なデータ解析ソリューションを提供することができました。オリジナルのハードウェアに依存しない実装と比較すると、新しい実装は、平均で 3 倍のパフォーマンス向上と最大 14 倍のスピードアップにより、膨大な量のデータ処理とモデル化を素早く正確に解析でき、顧客はデータの価値を迅速に発見・利用できるようになりました。

ソリューション : Tempo* ビッグデータ解析プラットフォーム

インテル® MKL とインテル® DAAL をベースに、インテルは MeritData と協力して、インテル® アーキテクチャー上でコア・アルゴリズム・ライブラリーを高速化する Tempo* ビッグデータ解析プラットフォームの作成に取り組みました。クラウド・コンピューティング・アーキテクチャーを利用することで、チームは高速なモデル化と解析を提供するビッグデータ解析ソリューションを実装しました。同時に、分野とデータ解析レベルの異なる顧客が、データの価値、データ可視化、詳細な解析を達成できるように、統合サービスの提供を実現しました。

Tempo* プラットフォームのシステム・アーキテクチャーには、データ・アクセス・レイヤー、解析とモデル化、結果表示、アクセスレイヤーが含まれており、統合されたクラウド・サービス・アクセス、クラウド・リソース・スケジューリング、クラウド・プラットフォーム管理を提供します。

BLOG HIGHLIGHTS

明示的なコンピューティングから暗黙的なコンピューティングへの転換

BOB DUFFY >

以前に、「Developers Need to Consider Relative Input For Intel® RealSense™ Technology」という記事で、コンピューティング分野で現在起こっている転換について触れました。インテル® RealSense™ テクノロジーを使用するコンピューター制御は、絶対制御よりも相対制御に最適です。これは、明示的なコンピューティングから暗黙的なコンピューティングへの転換と言えます。

明示的なコンピューティングとは?

明示的なコンピューティングとは、PC の時代 (1970 年代) から今日まで使用されているものです。コンピューター、ビデオゲーム、スマートフォンで何かしたい場合、デバイスに直接指示する必要があります。乗り物を移動するためジョイスティックを動かしたり、コンピューターにコマンドを送信するためキーボードで文字をタイプしたり、日付を選択するため左右にスワイプします。常に、アクションは明示的で、結果は予測可能です。

この記事の続きはこちら (英語) でご覧になれます。 >

データ・アクセス・レイヤー

データ・アクセス・レイヤーは、SQL/ 非 SQL データベース、Kafka* とのドッキング、Flume ストリーミング・データソース、非構造化テキスト・データソースを含む異なるデータソースを対応するために必要です。

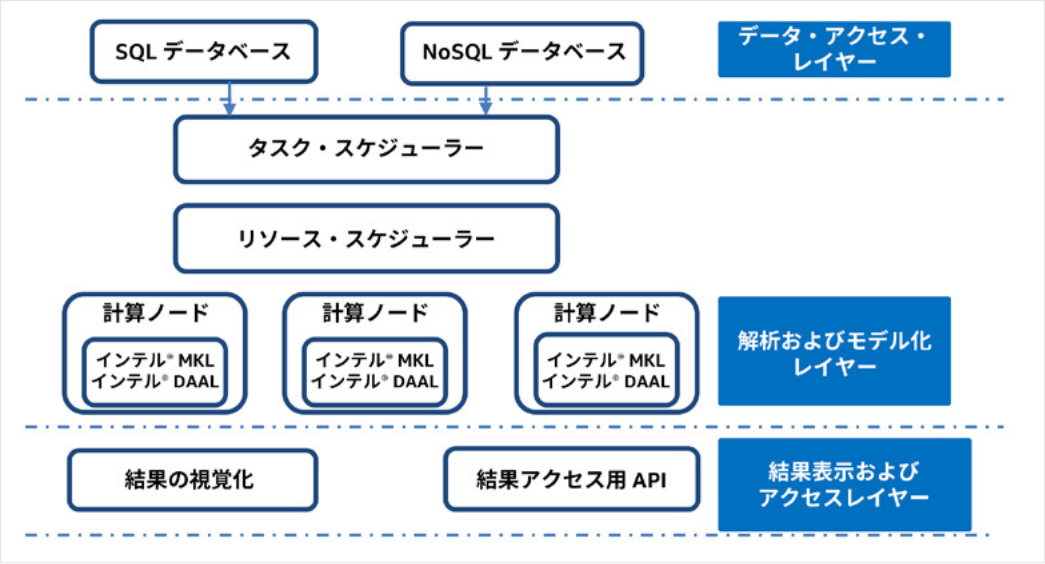
解析とモデル化レイヤー

インテル® MKL とインテル® DAAL をベースに、Tempo* データ解析プラットフォームのコア解析アルゴリズムを高速化します。インテリジェント・プラットフォームは、次の機能を提供します。

- ノード・スケジューリングの管理
- 割り当てられた計算ノードの解析
- 関連ジョブのログの記録
- 例外情報レコードの分類

結果表示とアクセスレイヤー

解析のモデル化が完了すると、モデル化の結果が統計として表示され、リソースの使用率を向上し、より素早い意思決定に役立てるため、結果レポートを生成できます。また、新しいデータソースを直接受け取り、正確な予測を行うため、結果を基に顧客が新しいモデルを作成することもできます。このレイヤーは、顧客が将来開発しやすいように、アクセス・インターフェイスとして API 呼び出しを提供します。



1 データマイニング・システム・アーキテクチャー向けの Tempo* ビッグデータ解析プラットフォーム

技術的な利点

インテルは、MeritData のニーズに応えるため、強力な CPU だけでなく、ハイパフォーマンスなソフトウェアも提供します。インテル® Xeon® プロセッサー E5-2699 V4 2 ソケット @ 2.20GHz は、最大 44 コア、88 スレッド、単精度で 3.0 TFLOPS のピーク・パフォーマンスをもたらします。仮想化やビッグデータ・マイニングのようなビジネスに不可欠なアプリケーションに最適です。

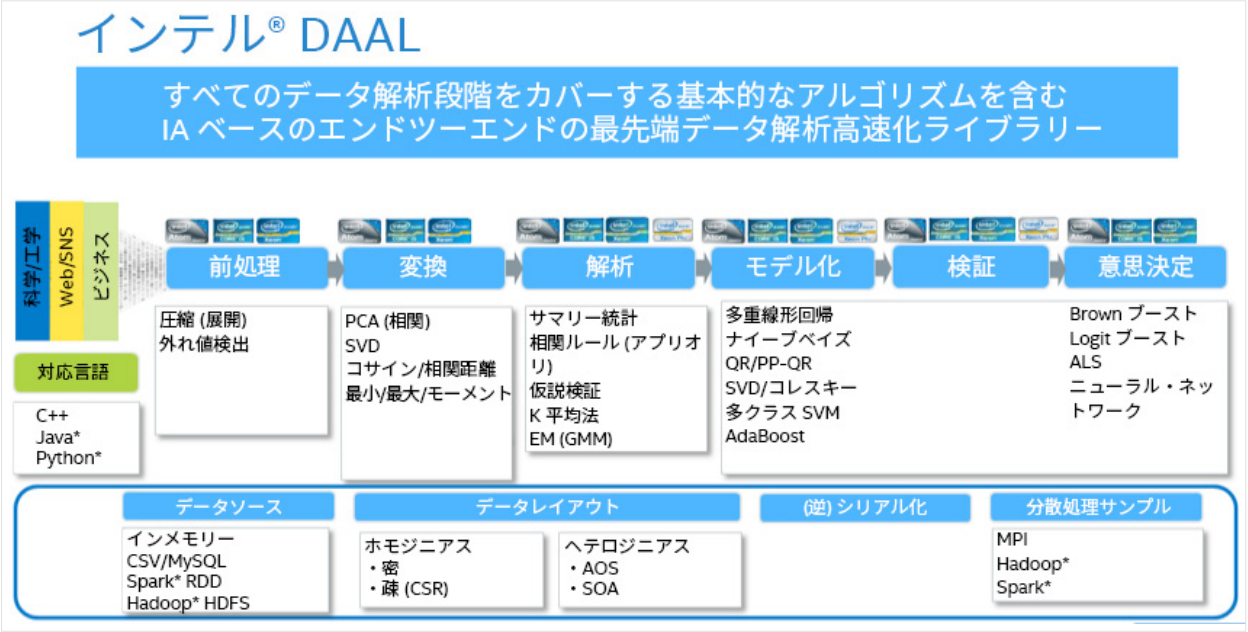
インテル® MKL は、インテル® アーキテクチャー上の複数レベルの並列処理向けに高度に最適化されている幅広い行列、ベクトル、演算処理ルーチンを提供します。これらのルーチンは、マルチコア、メニーコア、クラスター・アーキテクチャーで利用可能なリソースを効率良く活用し、自動的にインテル® Xeon® プロセッサーのワークロードのバランスを調整します。高度に最適化された関数 **(図 2)** は、高い計算パフォーマンスが求められる科学、工学、金融系アプリケーションで広く利用されています。これらのアプリケーションは、インテルのマルチコア・プロセッサーを最大限に利用して、アプリケーションのパフォーマンスを最大化し、開発期間を短縮することができます。

インテル® MKL には、BLAS/LAPACK/FFTW 関数が含まれています。これらの関数は、MATLAB* や NumPy* のようなサードパーティー・ソフトウェアとの統合を容易にします。また、Eigen のようなサードパーティー・ソフトウェアの対応する関数を簡単に置換できます。(詳細は**こちら**を参照してください。)

2 インテル® MKL のコンポーネント

インテル® MKL と同様に、インテル® DAAL もまたハイパフォーマンス・ライブラリーです。ただし、インテル® DAAL のほうがインテル® MKL よりもビッグデータ向けの機能が充実しています。オフライン、ストリーミング、分散解析のすべてのデータ解析段階 (前処理、変換、解析、モデル化、検証、意思決定) に対応した高度に最適化されたアルゴリズムのビルディング・ブロックにより、ビッグデータ解析の高速化を支援します。Hadoop*、Spark*、**R**、MATLAB* を含む主要データプラットフォームで非常に効率良いデータアクセスを提供するように設計されています。

インテル® DAAL **(図 3)** は、データセットの基本的な記述統計から高度なデータマイニングおよび**マシンラーニング**にわたる、豊富なアルゴリズムを提供します。ビッグデータ開発者が、比較的少ない労力で、多くのビッグデータ・アルゴリズム向けに高度に最適化されたコードを開発できるように支援します。



3 インテル® DAAL のコンポーネント

Tempo* パフォーマンスの向上

MeritData の顧客は、より大きなデータを、より高速に、より忠実にモデル化および解析できるように、常により高速な計算パフォーマンスを求めています。インテル® MKL とインテル® DAAL をベースに、インテルは MeritData と協力して、Tempo* 大規模データ解析プラットフォームのコア・アルゴリズムの高速化に取り組みました。その結果、オリジナルの実装と比較してパフォーマンスが大幅に向上し、顧客がデータ価値を最大化し、ビジネスニーズを満たし、膨大な量の新しいデータを素早く解析できるように効果的に対処することができました。

特に、Tempo* の ELM と内製 L1/2 スパース反復アルゴリズムを最適化できたことは大きな成果でした。

ELM アルゴリズム

ELM は、MeritData のデータマイニング・アルゴリズムの 1 つです。ELM アルゴリズムのコードをプロファイルしたところ、パフォーマンスのボトルネックは、行列計算 (行列や行列乗算のような関数の反転を含む) であることが分かりました。MeritData のオリジナルの実装では、Eigen ライブラリーを使用していました。Eigen ライブラリーはさまざまな行列演算を提供しますが、インテル® MKL と比較するとパフォーマンスが最適化されていません。MeritData は、関連する関数をインテル® MKL 関数に置換することにしました。コードの置換は簡単に行うことができました **(表 1)**。

オリジナルバージョン

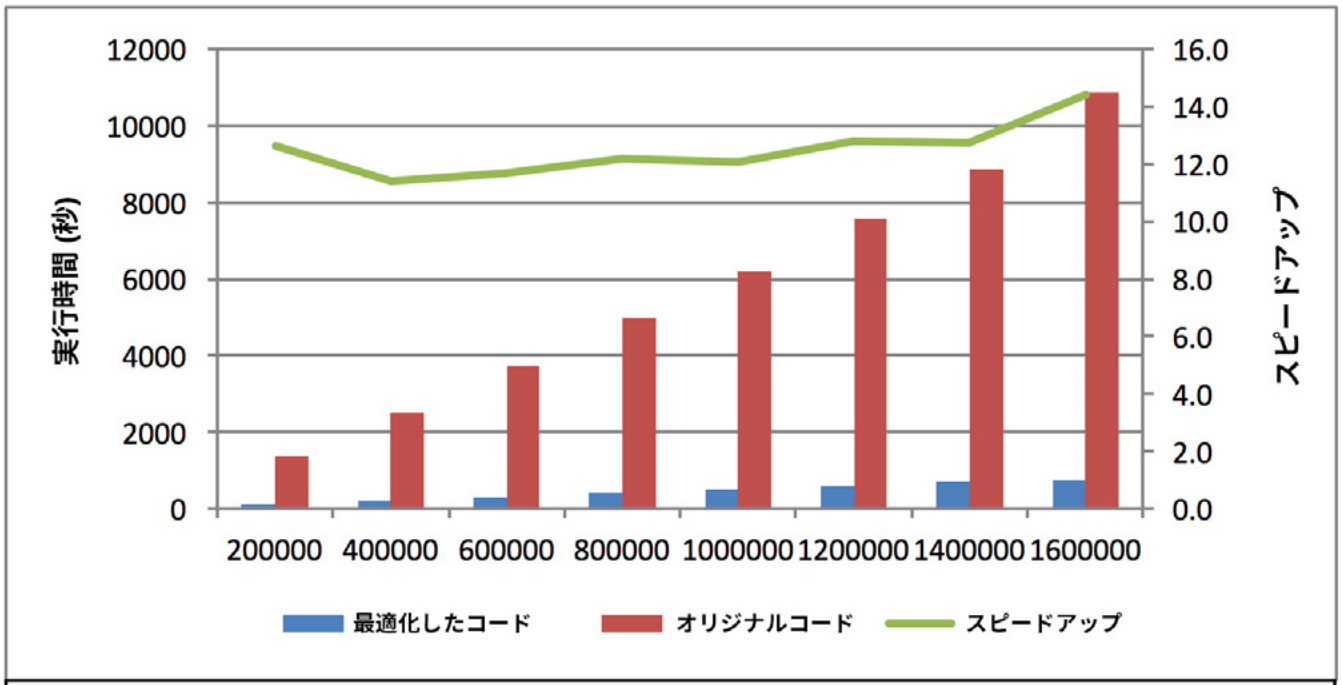
```
Eigen::MatrixXd Pk;
Eigen::MatrixXd Bk;
jobjectArray     update(JNIEnv*
env,jobject,jobjectArray
HM,jobjectArray TM){
// データの行番号を取得
int height = env->GetArrayLength(HM);
// データの列番号を取得
…
int width = env->GetArrayLength(tmp);
// 内部行列
Eigen::MatrixXd mat(height,width);
…
// TM 行列
Eigen::MatrixXd Tm(row2,col2);
…
// 計算を実行
Eigen::MatrixXd     matTran     =
mat.transpose();
Eigen::MatrixXd matTmp = matTran*mat;
...
// 逆行列
Pk = matTmp.inverse();
Bk = Pk*matTran*Tm;
…
}
```

インテル® MKL による最適化

```
double* Pk;
double* Bk;
jobjectArray update(JNIEnv*
env,jobject,jobjectArray HM,jobjectArray TM){
// データの行番号を取得
int rowNum = env->GetArrayLength(HM);
// データの列番号を取得
…
int colNum = env->GetArrayLength(tmp);
// 内部行列
double* mat = (double
*)mkl_calloc(rowNum*colNum, sizeof( double ),
64);…
// TM 行列
double* Tm = (double
*)mkl_calloc(rowNumT*colNumT,
sizeof( double ), 64); …
// 計算を実行
double* matTmp = (double
*)mkl_calloc(colNum*colNum, sizeof( double ), 64);
MKL_INT lda = colNum,ldc = colNum;
cblas_dgemm(CblasRowMajor,CblasTrans,CblasNoT
rans,colNum,colNum,rowNum,1.0,mat,lda,mat,co
Num,0.0,matTmp,ldc);
...
// 逆行列
MKL_INT* ipiv = (int *)mkl_calloc(colNum,
sizeof( int ), 32);
MKL_INT info =
LAPACKE_dgetrf(LAPACK_ROW_MAJOR,colNum,colNum
,matTmp,lda,ipiv);
info =
LAPACKE_dgetri(LAPACK_ROW_MAJOR,colNum,matTmp
,lda,ipiv);
…
}
```

表 1. マシンラーニング・アルゴリズムを利用して最適化する前と後のコード

インテル® Xeon® プロセッサー上で異なるサイズのデータを利用してテストしたところ、**図 4** に示す結果が得られました。最適化したアルゴリズムのパフォーマンスは平均で約 12 倍、最大 14 倍も向上しました。



システム構成: バージョン: インテル® MKL 11.3.2。ハードウェア: インテル® Xeon® プロセッサー E5-2699 v3 2.30GHz、2 ソケット x 18 コア、インテル® AVX2 対応、45MB キャッシュ、128GB メモリー。オペレーティング・システム: CentOS* 6.7。4 ノードクラスター、Spark* 1.5.1、Hadoop* 2.6.0、jdk-7u79-linux-x64、Scala 2.10.4 。ベンチマーク・ソース: MeritData のテストコードおよびテスト・データセット。

性能に関するテストや評価は、特定のコンピューター・システム、コンポーネント、またはそれらを組み合わせて行ったものであり、このテストによるインテル製品の性能の概算の値を表しているものです。システム・ハードウェア、ソフトウェアの設計、構成などの違いにより、実際の性能は掲載された性能テストや評価とは異なる場合があります。システムやコンポーネントの購入を検討される場合は、ほかの情報も参考にして、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。インテル製品の性能評価についてさらに詳しい情報をお知りになりたい場合は、http://www.intel.com/performance (英語) を参照してください。

* 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。

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

4 インテル® MKL により最適化する前と後の実行時間の比較

内製の L1/2 スパース反復アルゴリズム

L1/2 ベースのスパース反復アルゴリズムは、MeritData の内製アルゴリズムです。(当初、**Flanagan Java* 科学計算ライブラリー**を使用してコードを実装しました。) テストと解析を行ったところ、このアルゴリズムのパフォーマンス・ボトルネックは、カーネル関数と基本行列演算であることが分かりました。基本行列演算にはインテル® MKL の BLAS 関数を、カーネル関数にはインテル® DAAL のカーネル関数を使用するように変更しました。**表 2** は、オリジナルのコードとインテル® DAAL を使用するように変更した後のコードです。

オリジナルのコード

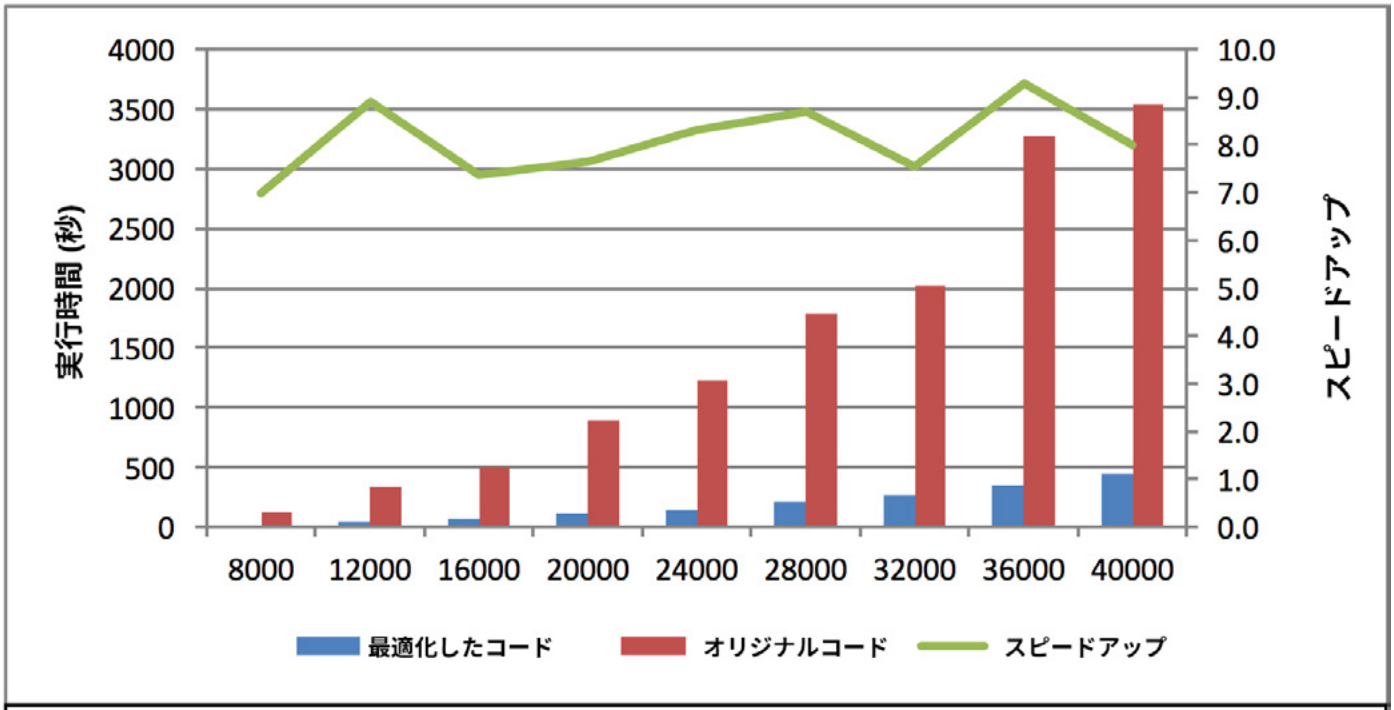
```
public Matrix getKernelMatrix()
throws Exception {
Matrix result = new
Matrix(m_data.numInstances(),
m_data.numInstances(), 0);
for (int i = 0; i <
m_data.numInstances() - 1; i++) {
for (int j = i + 1; j <
m_data.numInstances(); j++) {
result.set(i, j, evaluate(i, j,
m_data.instance(0)));
}
}
result = result.plus(
result.transpose().plus(
Matrix.identity(m_data.numInstances()
,
m_data.numInstances()))).copy();
return result;
}
```

インテル® DAAL による最適化

```
jobject getKernelMatrix(JNIEnv*
env,jobject,jdouble param,jint rows,jint
cols,jobject byteBuffer,jobject dstBuffer){
...
kernel_function::linear::Batch<>
linearKernel;
/* カーネル・アルゴリズムのパラメーターを設定 */
linearKernel.parameter.k = 1.0;
linearKernel.parameter.b = 1.0;
linearKernel.parameter.computationMode =
kernel_function::matrixMatrix;
/* アルゴリズムの入力データテーブルを設定 */
linearKernel.input.set(kernel_function::X, data);
linearKernel.input.set(kernel_function::Y, data);
/* 線形カーネル関数の計算 */
linearKernel.compute();
/* 計算結果の取得 */
services::SharedPtr<kernel_function::Result>
lkResult = linearKernel.getResult();
/* 結果の取得 */
services::SharedPtr<NumericTable> lkMat =
lkResult->get(kernel_function::values);
BlockDescriptor<double> block;
lkMat->getBlockOfRows(0, rows, readOnly,
block);
...
}
```

表 2. 反復アルゴリズムを最適化する前と後の L1/2 スパースコード

異なるサイズのデータで L1/2 スパース反復アルゴリズムをテストしたところ、最適化後のアルゴリズムのパフォーマンスは約 8 倍になりました **(図 5)**。



システム構成: バージョン: インテル® Parallel Studio 2017 Beta に含まれるインテル® DAAL。ハードウェア: インテル® Xeon® プロセッサー E5-2699 v3 2.30GHz、2 ソケット x 18 コア、インテル® AVX2 対応、45MB キャッシュ、128GB メモリー。オペレーティング・システム: CentOS* 6.7。ベンチマーク・ソース: MeritData のテストコードおよびテスト・データセット。

性能に関するテストや評価は、特定のコンピューター・システム、コンポーネント、またはそれらを組み合わせて行ったものであり、このテストによるインテル製品の性能の概算の値を表しているものです。システム・ハードウェア、ソフトウェアの設計、構成などの違いにより、実際の性能は掲載された性能テストや評価とは異なる場合があります。システムやコンポーネントの購入を検討される場合は、ほかの情報も参考にして、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。インテル製品の性能評価についてさらに詳しい情報をお知りになりたい場合は、http://www.intel.com/performance (英語) を参照してください。

* 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。

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

5 変更前と後の L1/2 スパース反復アルゴリズムの実行時間の比較

まとめ

データをより詳しく理解できるようにすることで MeritData の顧客は長年にわたって蓄積されたデータと新たに生成されるデータを統合し、限られたリソースでデータの価値を引き出したいと考えています。MeritData の Tempo* プラットフォームは、顧客のデータマイニング要求に対して、高速で効率的なソリューションを提供する必要があります。

インテルのエンジニアと緊密に協力して、MeritData はインテル® DAAL とインテル® MKL を利用して、Tempo* ビッグデータ解析プラットフォームのアルゴリズムの最適化に取り組みました。その結果、ビッグデータを高速に解析し、正確にモデル化し、分野とデータ解析レベルの異なる顧客を満足させることができるようになりました。ソリューションは、MeritData のデータ解析処理能力、パフォーマンス、ユーザー・エクスペリエンスを大幅に向上しました。

MeritData とインテルは今後も協力して、インテル® アーキテクチャーの計算能力の向上に対応するため、MeritData Tempo* の最適化に取り組んでいきます。この取り組みにより、MeritData の開発者は少ない労力で優れたパフォーマンスを達成し、顧客に最高のユーザー・エクスペリエンスを提供し続けることができます。

インテル® MKL の詳細 >

インテル® DAAL の詳細 >

Software

THE PARALLEL UNIVERSE