intel Software

THE PARALLEL UNIVERSE

生産性と C++ パフォーマンスの向上

インテル® C++ コンパイラー Standard Edition for Embedded Systems with Bi-Endian Technology

OpenMP* API バージョン 4.5: 標準化の進化

Issue 24 2016

目次

編集者からのメッセージ 4
北半球の春の訪れと時間の有効活用のヒント

James Reinders

注目記事

生産性と C++ パフォーマンスの向上 5
新しいインテル® SIMD Data Layout Template ライブラリーは、C++ コードを最適化して SIMD 効率の向上を支援します。

注目記事

インテル® C++ コンパイラー Standard Edition for Embedded Systems with Bi-Endian Technology 11
インテル® C++ コンパイラー Standard Edition for Embedded Systems with Bi-Endian Technology は、プラットフォームの制約を解消できるように開発者を支援します。

OpenMP* API バージョン 4.5: 標準化の進化 23
OpenMP* バージョン 4.5 は標準化の次のステップへの進化であり、オフロード・プログラミング向けの機能とともに新たな並列プログラミングのコンセプトを導入しています。

インテル® MPI ライブラリー: Hadoop* エコシステムのサポート 32
HPC 環境でのデータ解析が一般的になるにつれ、同じエコシステムでの MPI とビッグデータの使用が注目されるようになっています。

メモリーアクセスのパフォーマンス・ボトルネックの検出 42
新しいインテル® VTune™ Amplifier XE のメモリーアクセス解析機能を利用して困難なメモリー問題を解決する方法を紹介します。

インテル® IPP を利用した画像識別の最適化 62
インテルは、中国最大のインターネット・サービス・ポータルのエンジニアと協力して作業を行い、インテル® アーキテクチャー・ベースのプラットフォームでパフォーマンスを 2 倍に向上しました。

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

目次 (続き)

最新の IoT と組込みテクノロジーを利用したスマートな開発 73
インテル® マイクロコントローラー、IoT (Internet of Things) デバイス、組込みプラットフォームのコーディング、解析、デバッグ用のツールを紹介します。

インテルのクラスターツールを利用したハイブリッド・アプリケーションのチューニング 79
ハイブリッド・アプリケーションの分析とチューニングの詳細なワークフローを示します。

インテル® Advisor XE 2016 を利用したコードのベクトル化 89
ベクトル化アドバイザーには、次世代のインテル® Xeon Phi™ プロセッサー/コプロセッサーでのベクトル化を支援する新しい機能が搭載されています。

編集者からのメッセージ

James Reinders 並列プログラミングのエキスパート。
『**High Performance Parallel Programming Pearls**』(第 1 巻と第 2 巻)の共著者で、このほかに、『**Multithreading for Visual Effects**』(2014)、『**Intel® Xeon Phi™ Coprocessor High Performance Programming**』(2013、日本語:『**インテル® Xeon Phi™ コプロセッサー ハイパフォーマンス・プログラミング**』)、『**Structured Parallel Programming**』(2012、日本語:『**構造化並列プログラミング**』)、『**Intel® Threading Building Blocks: Outfitting C++ for Multicore Processor Parallelism**』(2007、日本語:『**インテル スレッディング・ビルディング・ブロック―マルチコア時代の C++ 並列プログラミング**』、中国語、韓国語の翻訳版があります)、『**VTune™ Performance Analyzer Essentials**』(2005) などの文献を発表しています。

北半球の春の訪れと時間の有効活用のヒント

いよいよ春がやってきました。庭で種蒔きをしていると春の訪れを実感できます。自然はにぎやかになり、日々の生活も忙しくなります。することは山のようにあるため、人々は生活をもっと楽にし、効率と生産性を高める方法を求めています。ソフトウェア開発には、単純で時間のかかる作業がつきものです。そのような退屈な作業を迅速に片付けることができれば、楽しいことにより多くの時間をかけることができます。

この号では、AOS から SOA への変換 (C++)、バイエンディアンのコンパイル、OpenMP* のサポートなど、さまざまなソリューションを利用して、できるだけ時間をかけずに、より優れたパフォーマンスを得る方法を紹介します。ベクトル化の作業において、AOS を SOA にすべきかのように、どのメモリーレイアウトが最高のパフォーマンスを達成できるか考えることは退屈でしょう。

1 つ目の注目記事、「**生産性と C++ パフォーマンスの向上**」では、新しいインテル® SIMD Data Layout Template (インテル® SDLT) を利用してメモリーレイアウトを最適化し、SIMD 対応の C++ コードのパフォーマンスを向上する方法を説明します。

ゆで卵に関するリリパットとブレフスキュの意見の違いを解決するツールは作成できませんが、元々ビッグエンディアン・プロセッサー用に書かれたコードをインテル® プロセッサー向けにコンパイルすることはできます。2 つ目の注目記事、「**インテル® C++ コンパイラー Standard Edition for Embedded Systems with Bi-Endian Technology**」では、このエディションのインテル® コンパイラーを利用して、ビッグエンディアン・アーキテクチャーからリトルエンディアン・アーキテクチャーへレガシー・アプリケーションを移行する方法を説明します。このツールを利用することで、コードを大幅に書き直すのではなく、再利用して両方のアーキテクチャーに対応することができます。

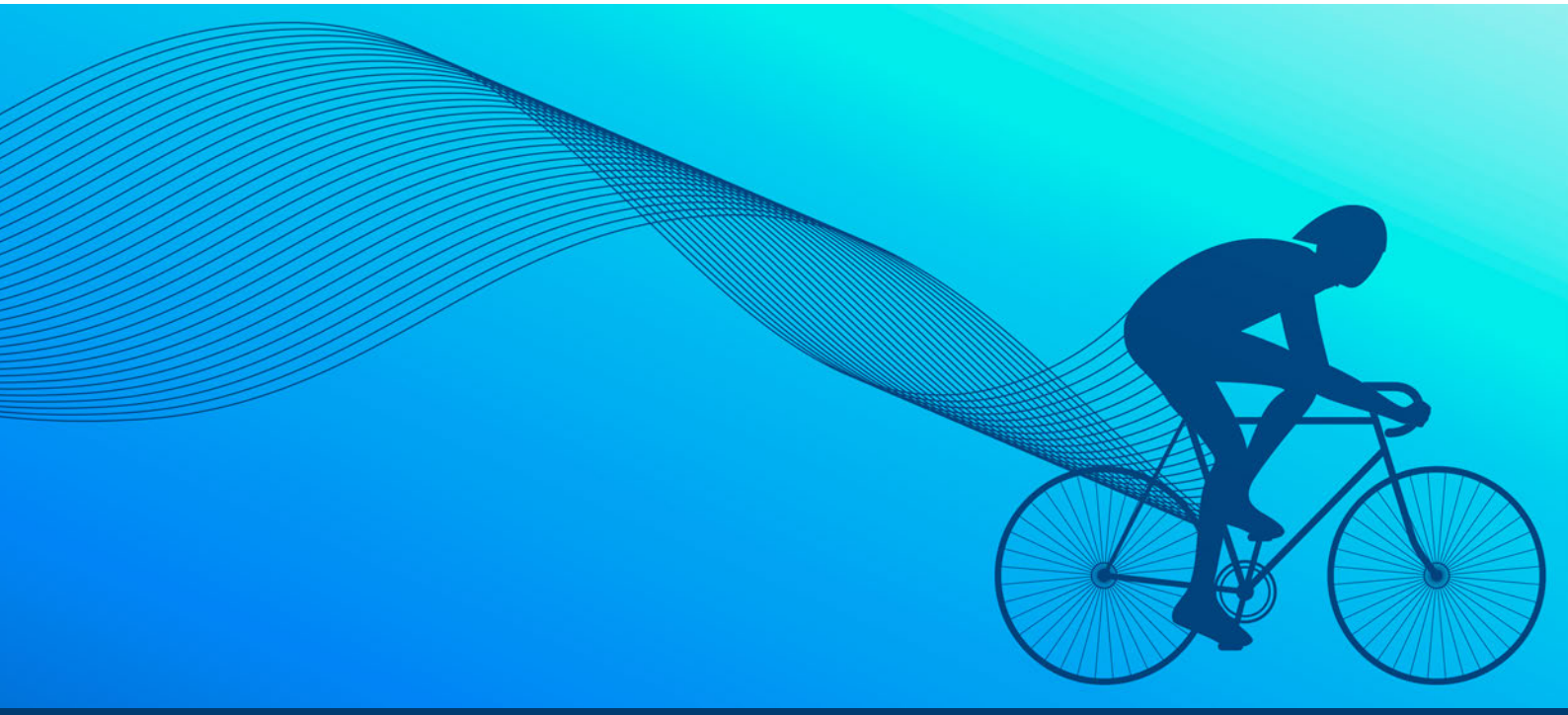
「**OpenMP* API バージョン 4.5: 標準化の進化**」では、OpenMP* の新しいバージョンの詳細と、アプリケーションとオフロードコードでより良い並列処理を表現する方法について説明します。OpenMP* 4.5 では、並列ループに起因する負荷バランスと構成の容易性の問題を解決してプログラマーの負担を軽減するタスク生成ループが追加されました。

日々の生活では常にやらなければいけないことがあるかもしれませんが、この号では、アプリケーションで高いパフォーマンスを追及する際に時間を節約できる方法をいくつか紹介しています。この節約できた時間をぜひ有効に活用してください!

James Reinders

2016 年 3 月

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



生産性と C++ パフォーマンスの向上

インテル® SIMD Data Layout Template (インテル® SDLT) でベクトル化された C++ コードの効率を高める

Udit Patidar インテル コーポレーション 開発製品部門 プロダクト・マーケティング・エンジニア

インテルでは、コードの hotspot を特定したり (インテル® VTune™ Amplifier XE など)、コードの最適化についてのアドバイスを表示する (インテル® Advisor XE など) さまざまなツールを提供しています。**インテル® C++ コンパイラー**は、特定の C++ ループの最適化が成功したかどうかを知らせる詳細な最適化レポートを生成します。[1] このたび、インテルは、インテル® Parallel Studio XE やインテル® System Studio のコンポーネントであるインテル® C++ コンパイラーに新しいテンプレート・ライブラリーを追加しました。C++ を使用しているアプリケーション開発者なら、メモリーアクセスのレイアウトを手動で最適化することがいかに大変であるかご存じでしょう。**インテル® SIMD Data Layout Template** (インテル® SDLT) ライブラリーは、プログラマーが最小の労力/オーバーヘッドで構造体配列 (AoS) 表現から配列

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

構造体 (SoA) 表現に変更できるようにして C++ コードを最適化します。SOA 表現は、メモリー使用量を抑え、データ構造レイアウトの変換を避けてベクトル化を容易にします。

最新のインテル® プロセッサーやコプロセッサーは、ベクトル命令に対応しており、SIMD (Single Instruction Multiple Data) プログラミング・モデルをサポートしています。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) 命令では、ベクトル化されたコードの浮動小数点演算のピーク・パフォーマンスは、ベクトル化されていないコードの 8 倍 (倍精度) または 16 倍 (単精度) になります。残念ながら、特に**ベクトル化**を考慮しないで書かれたレガシーコードでは、理論性能の限界に達することはまずありません。データのメモリーレイアウトを適切に行わなければ、多くのパフォーマンスは引き出せないままです。

C++ では、データのメモリーレイアウトが、効率的なベクトル化を達成できるかどうかを左右します。構造体および配列を扱う場合は特に重要です。開発者が **`std::vector`** のように C++ 標準テンプレート・ライブラリーのコンテナーで配列を表現することは一般的です。[2]

```
struct Point3s {

  float x;
  float y;
  float z;
};

std::vector<Point3s> inputDataSet(count);
std::vector<Point3s> outputDataSet(count);

for(int i=0; i < count; ++i) {
  Point3s inputElement = inputDataSet[i];
  Point3s result = // inputElement を変換するアルゴリズムに依存しないループ反復

  // アルゴリズムは高レベルのオブジェクトとヘルパーメソッドを使用
  outputDataSet[i] = result;
}
```

そのようなデータレイアウトは C++ プログラマーにとって当たり前の表現かもしれませんが、この AOS データセットをベクトルレジスターにロードするオーバーヘッドはベクトル化のパフォーマンス・ゲインを相殺してしまいます。AOS 表現よりも SOA 表現のほうがベクトル化に適していますが、C++ プログラマーの直観には反しています。[3]

インテル® SDLT の説明に移りましょう。

インテル® SDLT ライブラリーは、ユーザーには AOS インターフェイスを提供しますが、SOA 形式でデータをメモリーに格納します。プログラマーが **SIMD** に適したデータレイアウトに変換する必要はありません。インテル® SDLT は、標準 ISO C++11 機能を使用した高レベルのインターフェイスを提供します。特別なコンパイラーのサポートは必要ありません。インテル® SDLT のレイアウトは SIMD に適しているため、OpenMP* SIMD 拡張やインテル® Cilk™ Plus SIMD 拡張のようなインテル® コンパイラーのパフォーマンス機能を活用できます。

インテル® SDLT ライブラリーを使用するには、データ型をプリミティブとして宣言します。次に、(**`std::vector`** の代わりに) ジェネリックのインテル® SDLT コンテナーでプリミティブを使用します。配列ポインターやイテレーターの代わりに、コンテナーのデータアクセサーを使用します。明示的なベクトル・プログラミングで使用すると、インテル® SDLT コンテナーのアクセサーによりデータが変換され、コンパイラーにより効率的な SIMD コードが生成されます。

```
SDLT_PRIMITIVE(Point3s, x, y, z)

sdlt::soa1d_container<Point3s> inputDataSet(count);
sdlt::soa1d_container<Point3s> outputDataSet(count);

auto inputData = inputDataSet.const_access();
auto outputData = outputDataSet.access();

#pragma forceinline recursive
#pragma omp simd
for(int i=0; i < count; ++i) {
  Point3s inputElement = inputData[i];
  Point3s result = // inputElement を変換するアルゴリズムに依存しないループ反復

  // オブジェクト・ヘルパー・メソッドを使用してアルゴリズムを高レベルに保つ
  outputData[i] = result;
}
```

アクセサーを (ループ本体内の) ローカル変数へエクスポートする (またはローカル変数からインポートする) ループのインデックスとともに使用すると、コンパイラーのベクトライザーは SIMD に適したデータ形式にアクセスし、可能な場合はユニットストライド方式のロードを実行します。多くの場合、コンパイラーはループ内部のローカル・オブジェクトのプライベート表現を SOA に最適化することができます。上記の例では、コンテナーのメモリーレイアウトがコンパイラーのローカル・オブジェクトのプライベート表現と同様に SOA であるため、コンパイラーは効率的なユニットストライド方式のロードを生成することができます。

メモリーのデータを SOA で表現すると、コンパイラーは配列要素をベクトルレジスターに直接ロードできるようになります。その結果、SIMD ベクトル化の能力を引き出すことができます。対照的に、AOS レイアウトで直接計算を行うと、実行スロットを消費するにもかかわらず、結果に寄与しない余分な命令がベクトルレジスターを占有します。

要約すると、以下の表に示すように、インテル® SDLT を使用することで、最適なメモリーレイアウトが選択されて生産性が向上し、SIMD 対応 C++ コードのパフォーマンスが向上します。

C++ を使用している開発者の生産性を向上	C++ オブジェクトの使用を中止したり、C 配列に戻す代わりに、最小限の労力で済むジェネリック・コンテナーを使用してコードを SIMD 対応にします。データレイアウトの変換はインテル® SDLT に任せます。
パフォーマンスを向上	メモリーアクセスを連続させることにより、コンパイラーがより効率的な SIMD コードを生成できるようにします。また場合によっては、オーバーヘッドが大きいとされた個所でベクトル化が利用できるようになります。
統合が簡単	インテル® SDLT のコンテナーは、`std::vector` に似たインターフェイスを提供します。データアクセサーは、既存のインテルのベクトル・プログラミング・モデルと互換性があり、ほかのインテル® パフォーマンス・ライブラリーと調和します。

インテル® SDLT が適しているかどうか評価する場合は、次の言葉を考慮すると良いでしょう:[4]「ほとんどの場合、インテル® SIMD 対応のハードウェアではベクトル化しないよりもベクトル化するほうがよい」。

データレーン数が増加した最新のインテル® コプロセッサーでは、インテル® SDLT を考慮に入れないことによりコードの現代化が妨げられ、従来の C++ アプリケーションの欠点や短所が強調される可能性があります。C++ アプリケーションで SOA レイアウトを可能にする、新しいインテル® SDLT を試してみてください。SIMD の効率が向上し、思いがけない喜びが得られるでしょう。

インテル® SDLT が DreamWorks Animation の最先端の技術に貢献

AOS/SOA 問題は広く研究されています。そして、インテル® SDLT はすでに多くの業界で素晴らしい評価を得ています。例えば、DreamWorks Animation には、標準 C++ プログラミングの原理を利用して (つまり、明示的なベクトル化を考慮しないで) 記述された数学ライブラリーがあります。そのような大規模なコードのデータ構造レイアウトを手動で SOA に変換することは非常に困難です。数学ライブラリーはキャラクター・アニメーションのほぼすべてのエリアに影響するため、SIMD 対応の大きな障害となっていました。インテル® SDLT でコードを変換したところ、既存のコードを (変更なしで) コンパイルし、ループをベクトル化して、パフォーマンスを大きく向上できました。

「DreamWorks Animation では、[インテル® SDLT] を使用して社内アニメーション・ツール Premo* の変形コードをベクトル化しました。その結果達成できたパフォーマンスの向上は劇的なものでした。将来の映画に登場するキャラクターは、これまでになく高品質になることでしょう。ライブラリーは使いやすく、既存のコードベースにも簡単に統合することができました。」

DreamWorks Animation
Martin Watt 氏

サンプルコードとコンパイラーのドキュメント

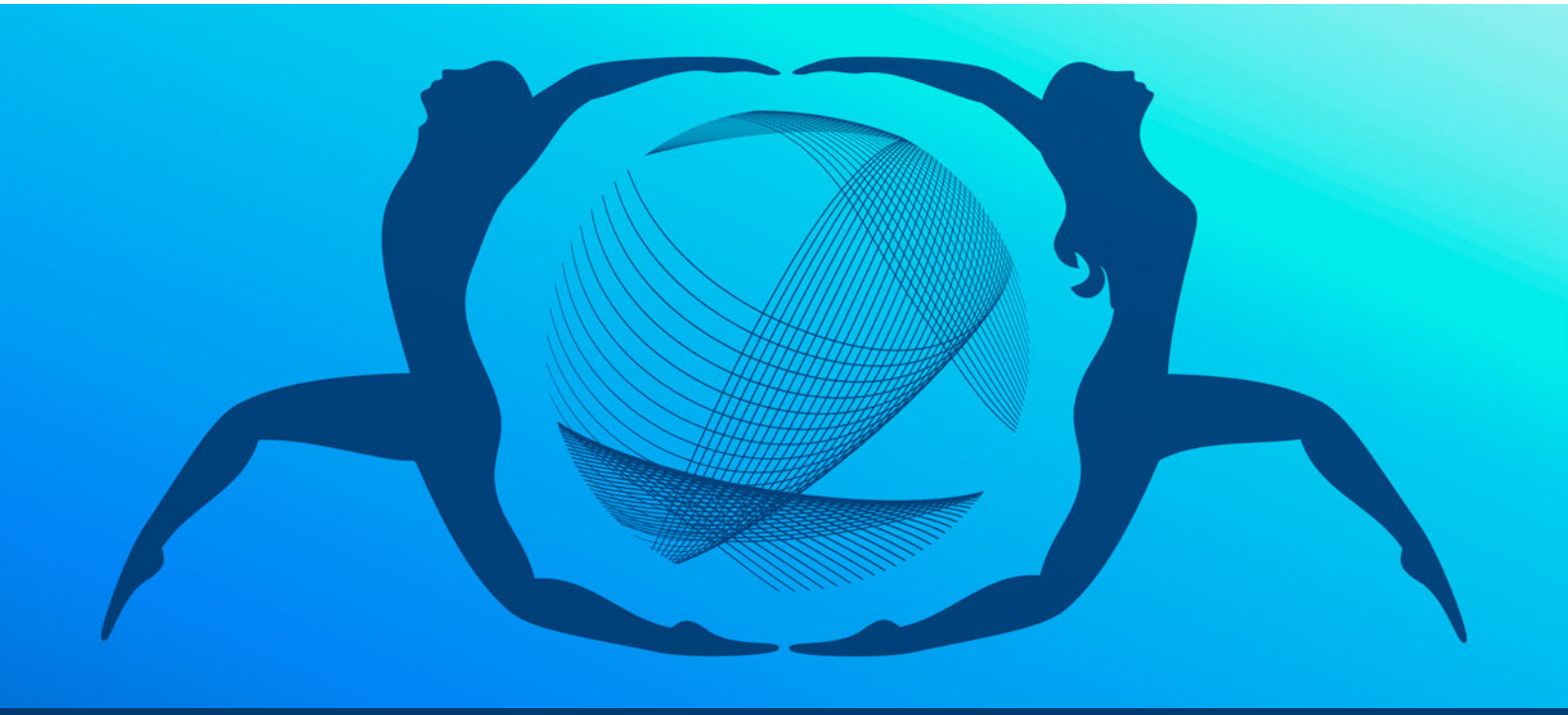
1. インテル® SDLT を使用した平均化フィルターの実装 (英語)
2. インテル® SDLT のサンプルコード (英語)
3. インテル® SDLT のサンプルコード (その 2) (英語)

参考文献

1. 「ベクトル化アドバイザーのヒントの活用」、The Parallel Universe Issue 23
2. インテル® SIMD Data Layout Templates (インテル® SDLT) 入門 (英語)
3. 32 ビット インテル® アーキテクチャーのメモリー利用を最適化するデータ構造の操作方法 (英語)
4. ケーススタディー: 計算負荷の高いループにおける構造体配列と配列構造体の比較

インテル® C++ コンパイラーの入手方法

インテル® Parallel Studio XE の 30 日間無料体験版のダウンロード >
インテル® System Studio の 30 日間無料体験版のダウンロード >
最寄りのリセラーを調べる >
学生、教育関係者、オープンソース貢献者: 無料版を入手する >



インテル® C++ コンパイラー Standard Edition for Embedded Systems with Bi-Endian Technology

ビッグエンディアン・プラットフォームとソフトウェアの依存性によるレガシー・プラットフォームの制約を回避してリトルエンディアン・アーキテクチャーへ簡単に移行する

Kittur Ganesh 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® C++ コンパイラー Standard Edition for Embedded Systems with Bi-Endian Technology は、ビッグエンディアン・ソフトウェアの依存性によりレガシー・プラットフォームを使い続けている開発者を支援するために開発されたツールで、レガシー・アプリケーションをビッグエンディアン・アーキテクチャーからリトルエンディアン・アーキテクチャーへ簡単に移行することが可能です。アプリケーションのコードベース全体を書き直すのではなく、最小限のコード変更でコードを再利用し、ビッグエンディアン・アーキテクチャーとリトルエンディアン・アーキテクチャーで共通のコードベースを利用して、スムーズに移行することができます。

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

インテル® C++ コンパイラー Standard Edition for Embedded Systems with Bi-Endian Technology は、インテル® C++ コンパイラーの長所を活用して、最高のアプリケーション・パフォーマンスを提供します。

この記事では、レガシー・アプリケーションをリトルエンディアン・アーキテクチャーのシステムへ移行するときに注意すべき問題を含む、主な機能の概要を説明します。

はじめに

1990 年代前半から中頃は、ビッグエンディアンの RISC アーキテクチャー (SPARC*、MIPS*、PowerPC* など) が組込み市場を支配しており、コンピューター・ネットワーキング、テレコミュニケーション、セットトップ・ボックス、DSL、ケーブルモデムなどで広く使用されていました。実際、ほとんどのアプリケーションはこれらのシステム向けに開発されていました。しかし、より高速でスケーラブルな新世代の x86 アーキテクチャー・プロセッサーがリリースされたことで、多くの開発者は、これらのアプリケーションをリトルエンディアンの x86 アーキテクチャー・システムに移植する最適な方法のジレンマに直面することになりました。

アプリケーションは標準化された高水準言語 (C/C++) で開発されていますが、従来のコンパイラーは単一エンディアンのアーキテクチャー向けにのみコンパイルすることができます。ソースコードにエンディアン・バイト順の依存性がある場合、ターゲット・アーキテクチャーで実行したときに重大なランタイム問題が発生するため、あるエンディアンのアーキテクチャーから別のエンディアンのアーキテクチャーへの移植は問題となります。これらの依存性に対応するには、開発者は発見が困難なバイト順依存コードを特定して、エンディアン中立コードまたはターゲット・アーキテクチャーのエンディアンに対応するコードに手動で変換する必要があります。

パフォーマンスのほうが市場投入までの時間 (TTM) よりも優先される場合、手動変換による移行は、ミスを引き起こしやすくコストを上昇させます。アプリケーションがすでにエンディアン中立の場合は、エンディアンがあいまいな EEMBC* ベンチマーク・サンプルのようなアプリケーションの再コンパイルのみ問題になります。

エンディアンの依存性を含むアプリケーションの場合は、**インテル® C++ コンパイラー** Standard Edition for Embedded Systems with Bi-Endian Technology を使用すると、優れたパフォーマンスを保ちながら、正しいコードを生成することができます。移植では、関数プロトタイプや変数宣言などのデータ・エンディアンの種類を調整し、手動による移植プロセスには、ビットフィールド・アクセスやシフト操作などを扱う煩わしい作業が含まれます。

通常のインテル® C++ コンパイラーと異なり、インテル® C++ コンパイラー Standard Edition for Embedded Systems with Bi-Endian Technology は、コンパイラー使用モデルの一部である言語拡張 (後述) により依存データのエンディアン・バイト順を指定することで、開発者がソースコード・ベースのバイト順でアプリケーションをコンパイルできます。コンパイラーは、適切なランタイム実行のため、必要に応じてバイト順依存コードにバイトスワップ命令を挿入して、メモリーにあるネイティブ・エンディアン・アーキテクチャーのデータがターゲット・エンディアン・アーキテクチャーのバイト順に変換されるようにします。

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

インテル® C++ コンパイラー Standard Edition for Embedded Systems with Bi-Endian Technology は、開発者がビッグエンディアン・アーキテクチャーからリトルエンディアン・アーキテクチャーへアプリケーションを移行できるように特別に設計された、Linux* をホストとし、Linux* をターゲットとする、コマンドライン・ベースのスタンドアロン・コンパイラーです。

次のような長所があります。

- **素早く簡単な移行。** インテル® アーキテクチャー (リトルエンディアン) へ素早く移行できます。
- **最小限のコード変更。** アプリケーションのコードベース全体を書き直すのではなく、最小限のコード変更によりビッグエンディアン・コードを再利用できるため、実装と検証にかかる労力が軽減されます。
- **共通のコードベース。** ビッグエンディアン・アーキテクチャーとリトルエンディアン・アーキテクチャーを 1 つのコードベースで管理できます。

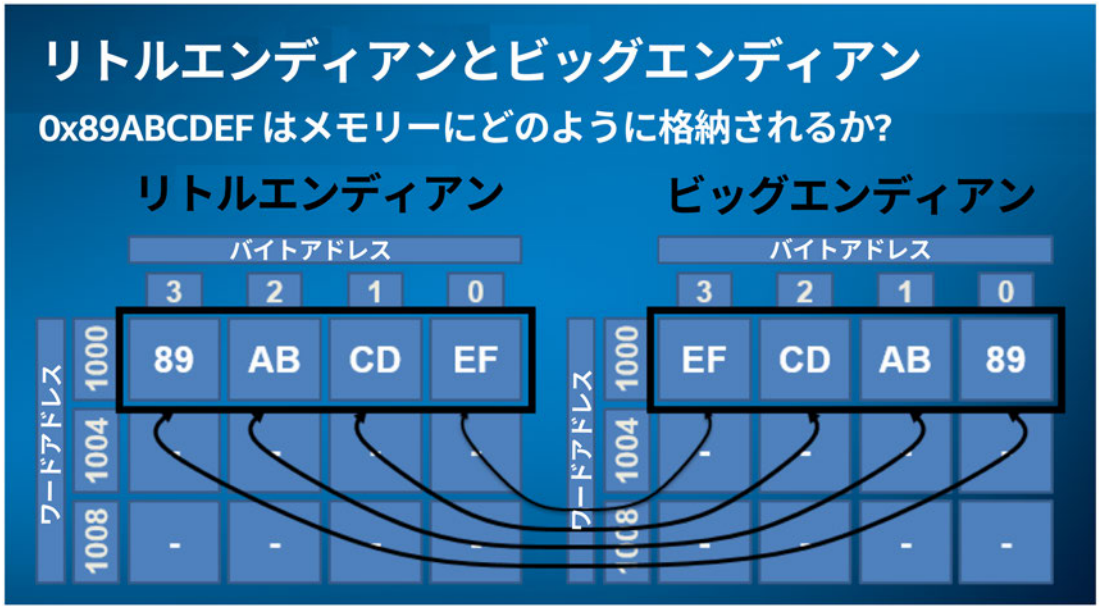
インテル® C++ コンパイラー Standard Edition for Embedded Systems with Bi-Endian Technology は、インテル® C/C++ コンパイラー for Linux* に似ており、共通の最適化とパフォーマンス・モデルに関する基本テクノロジーを使用しています。インテル® C/C++ コンパイラー for Linux* との大きな違いは、バイエンディアン・テクノロジーをサポートしていることです。

使用モデルの主な機能については、移植プロセスで注意すべき一般的な問題を含め、コード例を用いて後述します。主な機能を理解することは、インテル® C++ コンパイラー Standard Edition for Embedded Systems with Bi-Endian Technology を利用して、レガシー・アプリケーションをインテル® アーキテクチャーにシームレスに移行するのに役立つでしょう。

バイト・エンディアンのソリューションとバイト順の基本

バイト・エンディアンとは、マルチバイト・アクセスに関連するメモリーのデータレイアウト表現を指すシステム・アーキテクチャー特性です。リトルエンディアン変数は、最下位バイトがメモリー・バイト・アドレスの最下位に格納され、ビッグエンディアン変数は、最下位バイトがメモリー・バイト・アドレスの最上位に格納されます。ここでは、ビッグエンディアン・モデルは、メモリーでビッグエンディアンとリトルエンディアンの両方のデータを受け付けるものとします。

図 1 は、メモリー上のデータ 0x89ABCDEF を、リトルエンディアン変数とビッグエンディアン変数で表現しています。ビッグエンディアン変数では、リトルエンディアン変数と逆のバイト順を使用し、最下位バイト (EF) がメモリー・バイト・アドレスの最上位に格納されることが分かります。



1 リトルエンディアンとビッグエンディアン

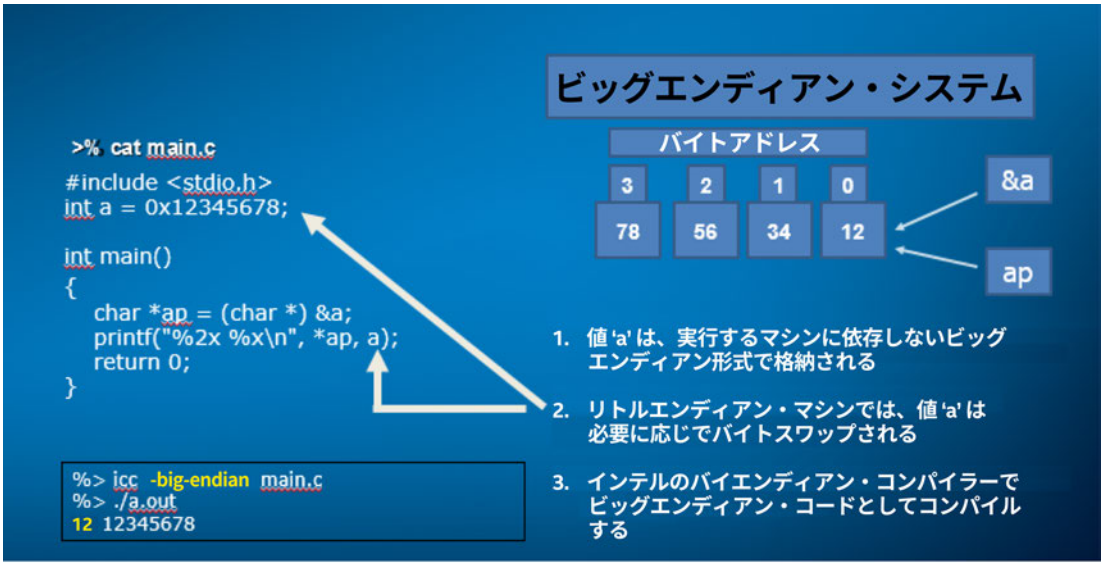
図 2 は、ビッグエンディアンとリトルエンディアン・アーキテクチャーのシステムで異なる結果を生成する、バイト順の依存性を含むサンプルコードです。ポインター ap は、リトルエンディアンのシステムでは最下位バイトの 78 を指し、ビッグエンディアン・システムでは最上位バイトの 12 を指します。この場合、プログラマーは、バイト順依存コードをエンディアン中立コードまたはターゲット・アーキテクチャーのエンディアンに対応するコードに手動で変換することができます。しかし、手動プロセスでは、レガシー・アプリケーションの数百万行のコードの中からバイト順依存コードを識別しなければならず、膨大な時間と労力が必要であり、ミスも引き起こしやすくなります。

図 3 は、インテル® C++ コンパイラー Standard Edition for Embedded Systems with Bi-Endian Technology による **図 2** のエンディアンの依存性問題に対するソリューションです。プログラマーがしなければならないことは、コンパイラーがビッグエンディアン・バイト順のセマンティクスを使用し、どちらのアーキテクチャーでも同じ結果を生成できるように、言語構造 (-big-endian オプション) を使用してデータのバイト順を識別することだけです。

図 3 のコードで、値「a」は、コードを実行するシステムに依存しないビッグエンディアン形式で格納されます。リトルエンディアンのシステムでは、値「a」は必要なときにコンパイラーによりバイトスワップされます。インテル® C++ コンパイラー Standard Edition for Embedded Systems with Bi-Endian Technology は、パフォーマンス・オーバーヘッドを抑え、生成される命令を最小限にするため、必要な場合にのみビッグエンディアンをエミュレートするバイトスワップ (bswap) 命令を挿入します。

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

2 バイト順に依存したコード実行



3 ビッグエンディアン・システムとリトルエンディアン・システムで同一の結果

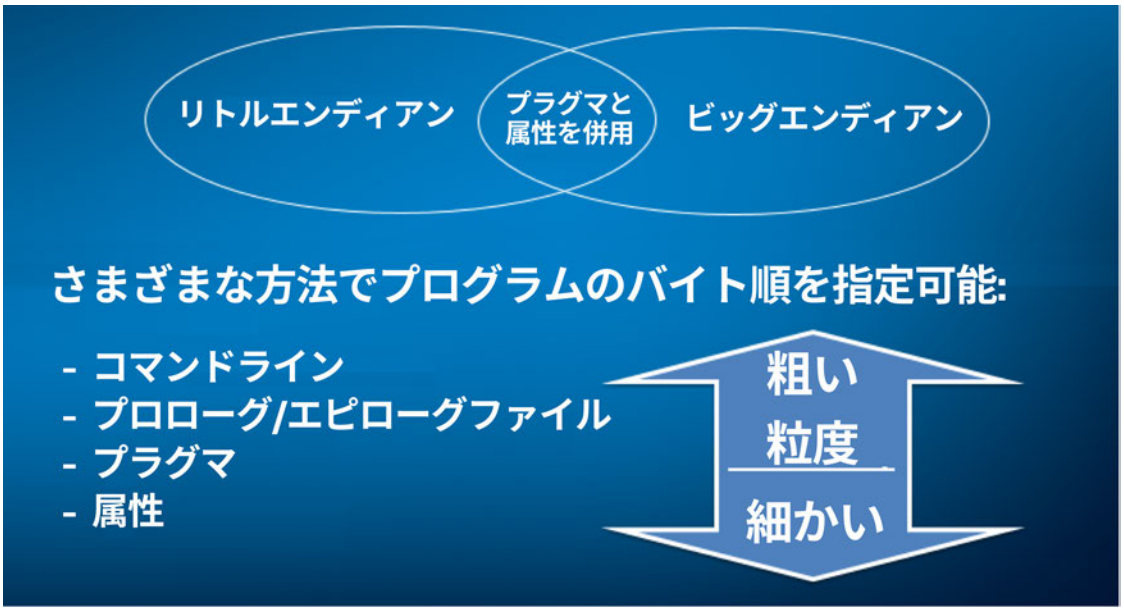
使用モデルと言語サポート

インテル® C++ コンパイラー Standard Edition for Embedded Systems with Bi-Endian Technology は、ビルトイン型や typedef、コード領域、変数宣言などのデータのバイト順 (型属性) を指定できるように、バイエンディアン言語拡張を提供します。これらの言語拡張は、ターゲット・アーキテクチャーでの実行時に、指定されたデータ型で適切なバイト順のセマンティクスを強制します。

図 4 に示すように、インテル® C++ コンパイラー Standard Edition for Embedded Systems with Bi-Endian Technology では、コマンドライン・オプション、プロローグ/エピローグ・ファイル・メカニズム、プラグマ、属性サポートなど、さまざまな方法でプログラムコードのバイト順を指定できます。

インテル® C++ コンパイラー Standard Edition for Embedded Systems with Bi-Endian Technology は、明示的または暗黙的な使用モデルによるビッグエンディアンあるいはリトルエンディアン変換を使用してデータアクセスを解決します。暗黙的なモデルは、-big-endian や -little-endian のようなコマンドライン・オプションにより、ファイルまたはアプリケーション全体のバイト順を指定したり、プロローグ/エピローグ・ファイル・メカニズムを使用して、ファイル全体またはディレクトリーで暗黙のエンディアン・モードを設定します。移行プロセスを単純にし、迅速に行えるように、できるだけ暗黙的なモデルを使用することを推奨します。

明示的なモデルは、pragma byte-order と属性によりエンディアンが混在した開発をサポートします。プログラマーは、さまざまな粒度で変数と関数のバイト順を指定できます。粒度を柔軟に指定できるため、明示的なモデルではエンディアンが混在した開発が可能です。



4 エンディアンが混在したコードの開発

図 5 は、プロローグ/エピローグオプションを使用してディレクトリーのエンディアン・バイト順を指定するサンプルコードです。また、**図 6** は、明示的な宣言により、より細かい粒度でコマンドライン・オプション -big-endian、#pragma byte_order、および属性を使用するサンプルコードです。コマンドライン・オプション -big-endian は、より細かい粒度で、プログラム全体のエンディアン順をビッグエンディアン・バイト順に設定します。#pragma byte_order (push, little-endian) は、# pragma 以降のインクルード・ファイルを含むすべての宣言にリトルエンディアン・バイト順を指定します。ただし、エンディアン属性によりビッグエンディアンと指定されている変数 b は除きます。同様に、float、配列、構造体、ビットフィールドを含む構造体などの異なるデータ型にバイト順を割り当てることもできます。

- **"/le" で終わるディレクトリーはリトルエンディアン**
  -pe-udir-rule=".*/le$ /usr/include/le-prolog.h /usr/include/le-epilog.h"
- **/proj1/ および /proj2/ ディレクトリーはビッグエンディアン**
  -pe-udir-rule="^/(proj1/|proj2/...) /usr/include/be-prolog.h /usr/include/be-epilog.h"
- **その他はすべてリトルエンディアン**
  -pe-idir-rule="^/usr/include/le-prolog.h /usr/include/le-epilog.h"

5 プロローグとエピローグ機能を使用した暗黙的なモード

一般に、インテル® C++ コンパイラー Standard Edition for Embedded Systems with Bi-Endian Technology のバイエンディアン機能は、次のように要約できます。コードがビッグエンディアンと指定された場合、コンパイラーは、あたかもそのコードがビッグエンディアン・システムで実行されるかのように扱い、必要な場所にバイトスワップ命令 (bswap) を挿入します。また、パフォーマンスを向上するため、排除可能なコードを取り除きます。同様に、リトルエンディアンと指定されたコードはネイティブコードとして実行され、bswap 命令は不要なため挿入されません。プログラマーは、アプリケーションでリトルエンディアンとビッグエンディアンのコードを混在させることができます。

図 6 は、互換性のないエンディアン・データ型 (msg) の不一致を含むサンプルコードです。この問題を解決するには、-symcheck オプションを指定して **`bepostld`** ツールを起動し、アプリケーションで使用されているすべてのシンボルの型チェックを実行します。型の不一致が見つかると、エラーメッセージとエラーの修正に役立つ詳細な情報が出力されます。**`bepostld`** ツールは、バイエンディアン・ポスト・リンク・ユーティリティーです。グローバル変数と関数の互換性のない型定義を検出したり、スタティック・ビッグエンディアン・ポインターの初期化を行います。

```
%cat main.c
#include <stdio.h>
 #pragma byte_order (littleendian)
int a = 0x12345678;
int __attribute__((bigendian))    b = 0x12345678;

int main()
{
  char* ap = (char*)&a;
  char* bp = (char*)&b;
  printf("%2x %x\n", *ap, a);
  printf("%2x %x\n", *bp, b);
  return 0;
}

%> icc -big-endian main.c
%> ./a.out
78 12345678
12 12345678
```

6 より細かい粒度でのバイト順属性の使用法

インテル® C++ コンパイラー Standard Edition for Embedded Systems with Bi-Endian Technology の採用

前述のように、1990 年代には、SPARC*、MIPS*、PowerPC* のようなビッグエンディアン・アーキテクチャー向けのアプリケーションが多く見られました。これらのレガシー・アプリケーションの多くは、エンディアンが混在する開発環境でまだ使用されています。インテル® C++ コンパイラー Standard Edition for Embedded Systems with Bi-Endian Technology は、インテル® アーキテクチャー・システムへの古いレガシーコードの移植を支援します。

図 7 は、そのような組込み環境における典型的な使用例です。

組込みオペレーティング・システム
- システム・ライブラリーおよび GNU* libc (リトルエンディアン)
- ハードウェア・インターフェイス (リトルエンディアン)
- ユーザー空間のルーティング・コード (ビッグエンディアン)
- バイエンディアン・インターフェイス (Linux* および libc)
  - 抽象化レイヤーを利用したシステム/libc 呼び出し
  - 使用されている libc およびシステムサービスを呼び出すためのラッパー (カスタマイズ)

ベアメタル
- 独自のオペレーティング・システムとルーティング・コード (ビッグエンディアン)
- ハードウェア・インターフェイス (リトルエンディアン)
- ハードウェア・インターフェイスを利用したバイエンディアン・インターフェイス

7 典型的な使用例

注意すべき主な問題

大規模なレガシー・アプリケーションは、ディレクトリーのエンディアンが正しく設定されていると、コンパイラーによるバイト順の不一致に関するすべての警告が排除され、**bepostld** ツールによるモジュール間のエンディアンの不一致エラーも排除され、関数プロトタイプが正しく実装されるため、移植が容易になります。以下は、移行プロセス中によくある問題です。

1. **プロトタイプの省略。図 8** は、プロトタイプの省略の例です。コンパイラーは、atoi() と printf() がリトルエンディアンであることを知らないため、バイトスワップした正しくない引数を渡します。そのため、このサンプルを実行すると、プログラムはクラッシュします。

   解決策: stdio.h および stdlib.h インクルード・ファイルを追加します。

```
%cat foo.c
int main() {
 char *ten = "10";
 int I = atoi(ten);
 printf("%d\n", i);
}

%> icc -big-endian -O0 foo.c
foo.c(3): warning #266: function declared implicitly
      int i = atoi(ten);
              ^
foo.c(4): warning #266: function declared implicitly
    printf("%d\n", i);
    ^
```

8 プロトタイプの省略

2. **誤ったキャスト。図 9** は、仮引数と実引数のサイズが異なる場合のサンプルコードです。

   解決策: -param-byte-order=little-endian を指定してコンパイルします。

```
#include <stdio.h>
typedef void (*fp_t) \
  (int __attribute__((bigendian)));
void foo(short __attribute__((bigendian)) arg) {
    printf("%d\n", arg);
}
int main() {
    int i = 10;
    fp_t fp = (fp_t)&foo;
    fp(i);
}
%> icc -big-endian  -O0 fp.c
%> a.out
```

9 誤ったキャスト

3. **ポイント先の型の不一致。図 10** は、sscanf() がリトルエンディアン値をビッグエンディアン変数 ‘i’ に代入しているため、型の不一致になります。

   解決策: ‘i’ をリトルエンディアンとして宣言し、‘sscanf’ のラッパー内部の値をスワップします。あるいは、-resolve-byte-order-mismatch を指定してコンパイルし、コンパイラーに自動的に問題を解決させることもできます。

```
// foo.c

#include <stdio.h>
int main() {
  int i = 1;
  sscanf("4", "%d", &i);
  printf("i=%d\n", i);
}
```

```
%> icc –big-endian foo.c
foo.c(4): warning #1872: byte order mismatch in format string conversion
    sscanf("4", "%d", &i);
                  ^

%> icc -big-endian -resolve-byte-order-mismatch foo.c
foo.c(3): remark #18014: Byte order of variable "i" was changed to little-
  endian to resolve mismatch during pointer conversion
```

10 ポイント先の型の不一致

4. **ポインター型のバイト順の省略。図 11** は、バイト順が指定されていないポインターを含むサンプルコードです。この問題を解決するには、**図 11** に示す属性を使用して、‘end’ ポインターをリトルエンディアンとして指定します。

```
// foo.c
#include <stdlib.h>
#include <stdio.h>
int main()
{
  char *str = "1.0 2.0";
  char *end;
  double d;
  d = strtod(str, &end);
  printf("%lf \"%s\"\n", d, end);
  return 0;
}
```

```
%> icc –big-endian foo.c
foo.c(11): warning #1696: implicit pointer conversion changes
byte order
    d = strtod(str, &end);
                    ^
次のように修正:
      char *end __attribute__((littleendian));
./a.out will produce:
1.000000 " 2.0"
```

11 ポインター型のバイト順の省略

5. **可変長引数。図 12** は可変長引数を含むサンプルコードです。vprintf() は vargs をリトルエンディアン・バイト順で読み取りますが、‘foo’ はビッグエンディアン引数で呼び出されるため、バイト順の不一致が発生します。コンパイラーは、va_list で渡される引数のバイト順を自動調整しません。また、可変長引数は同じバイト順で渡し、読み取るべきです (**図 13**)。

```
#include <stdio.h>
#include <stdarg.h>
void foo(const char *format, ...) {
    va_list args;
    va_start(args, format);
    //
    vprintf(format, args);
    va_end(args);
}
int main() {
    // args are passed in big-endian byte order.
    foo("%d %d\n", 1, 2);
}
%> icc -big-endian foo.c
main.c(7): warning #2333 : possible byte order mismatch passing va_list
    vprintf(format, args);
                    ^
```

12 可変長引数

```
#include <stdio.h>
#include <stdarg.h>
typedef int __attribute__((littleendian)) le_int;

#pragma byte_order(push, littleendian)
void foo(const char *format, ...) {
#pragma byte_order(pop)
    va_list args;
    va_start(args, format);
    printf(format, va_arg(args, le_int));
    va_end(args);
}
int main() {
    foo("%d\n", 1);
}
```

13 可変長引数 (続き)

6. **main()** 関数の引数とバイト順に関する注意点:

- デフォルトでは main の引数はリトルエンディアンとして扱われます。

```
typedef __attribute__((littleendian)) int leint;
int main( leint argc, char __attribute__((littleendian)) **argv )  { }
```

- ビッグエンディアン引数が必要な場合は次のように指定します。

```
typedef __attribute__((bigendian)) int beint;
typedef  __attribute__((bigendian)) char  *pIntel C++ Compiler Standard
    Edition for Embedded Systems with Bi-Endian Technologyhar;
int main (beint argc,  pIntel C++ Compiler Standard Edition for Embedded
    Systems with Bi-Endian Technologyhar *argv __attribute__((bigendian)) {}
```

注: ビッグエンディアン型がコードで明示的に設定されているため、インテル® C++ コンパイラー Standard Edition for Embedded Systems with Bi-Endian Technology は上記の診断や警告は行いません。

7. インテル® C++ コンパイラー Standard Edition for Embedded Systems with Bi-Endian Technology による移植のヒント:

- ファイルとディレクトリーの正しいエンディアンを保証します。
- バイト順の不一致に関する警告をすべて排除します。
- `bepostld` ユーティリティーによりレポートされたエラーを修正します。
- サイズの不一致 (16 と 32)、可変長引数、関数ポインター、型キャストなどにより発生するビッグエンディアンの問題に注意します。
- 適切なエンディアン・バイト順の正しい関数プロトタイプを保証します。

まとめ

インテル® C++ コンパイラー Standard Edition for Embedded Systems with Bi-Endian Technology は、ビッグエンディアン・ソフトウェアの依存性によるプラットフォームの制約を解消しようとしている開発者を支援するために開発されたツールで、大規模なレガシー・アプリケーションをリトルエンディアン・アーキテクチャーへ移行できます。ここでは、バイト順に依存するアプリケーションをビッグエンディアン・アーキテクチャーからリトルエンディアン・アーキテクチャーへ移行する際に注意すべきいくつかの問題について述べました。インテル® C++ コンパイラー Standard Edition for Embedded Systems with Bi-Endian Technology は、従来のインテル® C/C++ コンパイラー for Linux* と共通の最適化とパフォーマンス・モデルに関する基本テクノロジーを使用し、優れたパフォーマンスを実現します。

インテル® C++ コンパイラー Standard Edition for Embedded Systems with Bi-Endian Technology を評価する

詳細 >

OpenMP* バージョン 4.5: 標準化の進化

ループを容易に並列実行し異種プログラミングをサポート

Michael Klemm インテル コーポレーション シニア・アプリケーション・エンジニア
Christian Terboven アーヘン工科大学 HPC グループ

OpenMP* API 仕様はよく知られるようになり、共有メモリーシステムにおけるマルチスレッド化で広く利用されています。バージョン 4.5 はこの標準化の次のステップへの進化であり、オフロード・プログラミング向けの機能とともに新たな並列プログラミングのコンセプトを導入しています。

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

歴史

以前のバージョンの OpenMP* API 仕様 (4.0) は、2013 年 7 月に公開されました。このバージョンの主な機能追加は、ホストからコプロセッサーへのオフロード計算による異種プログラミングの基本機能がサポートされたことです。その後、ハイパフォーマンス・コンピューティングの分野で計算能力を高めるため、コプロセス (共同処理) に多くの関心が集まりました。これにより、OpenMP* 4.0 では異種コンピューティングをサポートする上で重要な機能を欠いていることが明らかになりました。また、プログラマーは常に新しい並列化の可能性とより優れた並列処理を記述することを求めています。

OpenMP* 4.5 では、コプロセッサーへのオフロード機能を改善するだけでなく、**並列プログラミング**を容易にする便利な機能を追加することに努めました。この記事では、OpenMP* 4.5 における 3 つの新機能について説明します。

- ループを簡単に並列実行するため、ループでタスクを生成します。
- ロックを使用する際に、ロックにヒントを追加することで並列性を高めます。
- システム上のコプロセッサーを使用するためオフロードを改善しました。

この記事の**ドイツ語版**は、*Heise Developer* オンラインマガジンに掲載されました。

タスク生成ループ

並列ループは、OpenMP* アプリケーションにおける最も重要な領域の 1 つです。これまでのワークシェア構文 (C/C++ の for 文、Fortran の do 文) は、ループ反復をチャンクに分割して並列チーム内の各スレッドにそれらを分配する、スレッド並列ループを表現する非常にシンプルな方式でした。しかしワークシェア構文には、大規模なアプリケーションのコードに対処する際に、並列プログラマーの作業を複雑にするいくつかの制約があります。最も重大な問題の 1 つは、ワークシェア構文の中に入れ子のワークシェア構文を記述することが OpenMP* で禁止されていることです。そのため、並列ループ内では、入れ子になった内部の並列ループを実行するには、新たにスレッドのチームを生成しなければいけません。

```
void taskloop_example() {
#pragma omp taskgroup
    {
#pragma omp task
        long_running_task()  // 同時に実行可能

#pragma omp taskloop collapse(2) grainsize(500) nogroup
        for (int i = 0; i < N; i++)
            for (int j = 0; j < M; j++)
                loop_body();
   }
}
```

1 新しい taskloop 構文で OpenMP* タスクを利用するコード例

新しい **taskloop** 構文は、ループのチャンクを直接ワーカースレッドに割り当てて実行する代わりに、OpenMP* タスクを使用することでこの問題を解決します。OpenMP* タスクは任意の階層に入れ子にできるため、ワークシェア構文の入れ子の制限は適用されません。**図 1** は、OpenMP* タスクと **taskloop** 構文を併用する方法を示します。**long_running_task()** 関数を OpenMP* タスクとして呼び出すことで、その長い実行時間を後続のループと同時に実行することができます。**taskloop** 構文は、ループ反復空間をチャンクに分割し、それぞれのチャンクに対して 1 つの OpenMP* タスクを生成します。処理を終えた任意のワーカースレッドが、それらのタスクをピックアップして実行します。長い時間を要するタスクが早期に終了した場合、その実行スレッドも **taskloop** によって生成されたタスクをピックアップすることができます。タスクを任意の階層に入れ子にできるため、実行中の各タスクで新しいタスクを生成して、アプリケーションの並行性をさらに高めることが可能です。OpenMP* ランタイムシステムは、すべてのタスクが実行を終えるまでワーカー全体のタスクの負荷バランスを調整します。

```
template<class K, class V>
struct hash_map {

  hash_map() {
    omp_init_lock(&lock);
  }

  ~hash_map() {
    omp_destroy_lock(&lock);
   }

  V& find(const K& key) const {
    V* ret = 0;
    omp_set_lock(lock);
    ret = internal_find(key);
    omp_unset_lock(lock);
    return *ret;
  }

  void insert(const K& key, const V& value)
{
    omp_set_lock(lock);
    internal_insert(key, value);
    omp_unset_lock(lock);
  }
  //...

private:
  mutable omp_lock_t lock;
  hash_buckets *buckets;
  // ...
};
```

2 キーを値にマップするハッシュマップの例

taskloop 構文は、ワークシェア構文とタスク構文の両方からシンタックスを継承します。データの属性 (**shared**、**private**、**firstprivate**、および **lastprivate**) を制御する通常の節に加えて、タスク節 (**final** と **mergeable**) をサポートしています。さらに、nogroup 節もサポートします。この節は、構文によって生成されたすべてのタスクを自動的に同期する、暗黙的なタスクのグループを非アクティブ化します。例では、長時間実行されるタスクと **taskloop** 構文の実行を同期するため、明示的な **taskgroup** 構文を使用しています。

生成されるタスクのサイズは、**grainsize** 節で制御できます。これは、生成されるタスクにループ反復をいくつ割り当てるか定義します。プログラマーがタスク数の指定を望む場合、代わりに **num_tasks** 節を使用できます。さらに、完全に入れ子になったループを collapse 節を使用して 1 つのループにまとめることができます。これは、ワークシェア構文における collapse 節の作用と同じです。

OpenMP* 4.5 では、OpenMP* タスク構文 **task** と **taskloop** に **priority** 節が追加されています。この節は、ランタイムシステムによるタスクの実行順序に影響します。priority 節には正の整数値を指定します。値が大きいほど生成されるタスクの優先順位は高くなります。低い優先順位のタスクの前に高い順位のタスクを実行する必要があることを、ランタイムシステムへ知らせるヒントとして使用されます。しかし、タスクの実行に際して特定の順番を完全に保証することはできないため、OpenMP* の実装ではヒントに固執する必要はありません。priority 節が省略されると、デフォルトでゼロ (0) が仮定されます。

Locks、Locks、Locks

ロックの取得と解除による相互排他は、大部分の並列プログラムで避けられない必要悪です。競合状態やデータ衝突を回避するため、共有リソースにアクセスするコード領域に入る前にロックを取得しなければいけません。ロックによるリソースの保護には、実行をシリアル化するため並列処理を制限するという代償が伴います。アプリケーションによっては、共有リソースへの同時アクセスが衝突する可能性が非常に低い (ゼロではない) にもかかわらず、安全性の理由からロックが利用されることがあります。

図 2 のコード例は、型 **v** の値に型 **K** のキーをマッピングするハッシュマップを単純化した非効率な実装です。エラー処理やほかの最適化を実装しない簡単な例ですが、このコードはロックに関する問題を抱えています。排他制御は、**hash_map** クラスの各メソッドに入った後に直ちにロックを取得することで有効になります。すると、個々のハッシュバケットの潜在的な競合状態を避けるため、実行がシリアル化されます。ハッシュ関数とデータ構造は、できるだけアクセス競合を回避するように設計されますが、これはときには不要なこともあります。各スレッドが異なるハッシュバケットや要素上で動作している場合、複数のスレッドがハッシュマップを操作するコード領域に入るのは安全であると考えられます。一般的な実装では、個々のバケットや要素を保護するロックを使用するか、ロックなしのデータ構造が採用されます。いずれの場合でも、スケーラブルかつ効率良いソリューションを見つけるため、プログラマーはかなりの作業を強いられます。

```
template<class K, class V>
struct hash_map {

  hash_map() {
    omp_init_lock_with_hint(&lock,
      kmp_lock_hint_hle);
  }

  ~hash_map() {
    omp_destroy_lock(&lock);
  }

  V& find(const K& key) const {
    V* ret = 0;
    omp_set_lock(lock);
    ret = internal_find(key);
    omp_unset_lock(lock);
    return *ret;
  }

  void insert(const K& key, const V& value)
{
    omp_set_lock(lock);
    internal_insert(key, value);
     omp_unset_lock(lock);
  }

private:
    mutable omp_lock_t lock;
    hash_buckets *buckets;
};
```

3 クリティカル領域を楽観的に実行するため、投機的なロックを使用

OpenMP* 4.5 は、この負担を軽減する新機能を提供します。プログラマーは、新しい API を使用して OpenMP* ランタイムにロックのヒントを渡すことで、アプリケーション・コード内のロックの使用目的を知らせることができます。ロックを初期化する 2 つの新しい関数、**omp_init_lock_with_hint** と **omp_init_nest_lock_with_hint** が定義されています。これらの関数は、型 **omp_lock_hint_t** の追加の引数を受け付けます (ロックのヒントについては、**表 1** をご覧ください)。ほかの OpenMP* のヒントと同様に、意図するロックの実装を最適化するためヒントを使用することができます。例えば、OpenMP* の実装は、テストとセットロックを futex (fast userspace mutex: 高速ユーザー空間 mutex の略) ベースのロックで置き換えることができます。また、特別なハードウェア命令 (例えば、インテル® トランザクショナル・シンクロナイゼーション・エクステンション: インテル® TSX) を利用することもできます。すべての状況において、プログラムの動作に影響を与えないようロックのセマンティクスは保持されます。

ヒントは整数式で表します。C/C++ では "|" オペレーターで、Fortran では "+" オペレーターで組み合わせることができます。これは、特定のロックを使用する際に、状況に応じた表現が可能となるため、プログラマーの柔軟性が高まります。OpenMP* 規格は、実装において事前定義されたヒントを追加拡張することを許可しています。インテルの OpenMP* ランタイムでは、**表 2** の追加のヒントが定義されています。

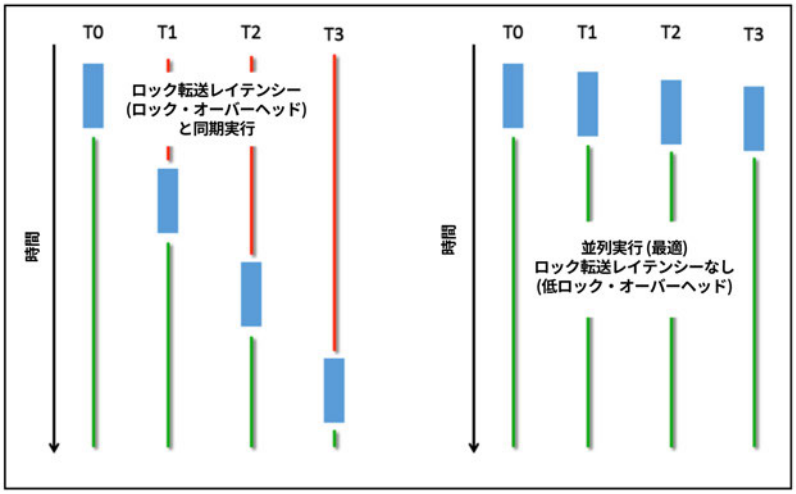
ヒント	セマンティクス
omp_lock_hint_none	OpenMP* ランタイムは、自由にロックの実装を選択できます。
omp_lock_hint_uncontended	スレッドが同時にロックにアクセスすることは少なく、競合の可能性は低いです。
omp_lock_hint_contended	複数のスレッドが同時にロックを取得しようとするため、頻繁に競合するロックを最適化します。
omp_lock_hint_ nonspeculative	楽観的なロックを使用しないことを指示します。取得するスレッドのワーキングセットがオーバーラップするため、かなりの競合があります。
omp_lock_hint_speculative	楽観的なロックを使用することを指示します。スレッドのワーキングセットは、ほとんど競合しないことが期待できます。

表 1. OpenMP* 4.5 でサポートされるロックのヒント

ヒント	セマンティクス
kmp_lock_hint_hle	インテル® TSX のハードウェア・ロック省略機能を使用します。
kmp_lock_hint_rtm	インテル® TSX の RTM 機能を実装したロックを使用します。
kmp_lock_hint_adaptive	競合をチェックしてあまりにも多くの衝突が発生する場合は、伝統的なテスト & セットロックへフォールバックする投機的な適応ロックを使用します。

表 2. インテルの OpenMP* ランタイムで定義される追加のロックのヒント

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



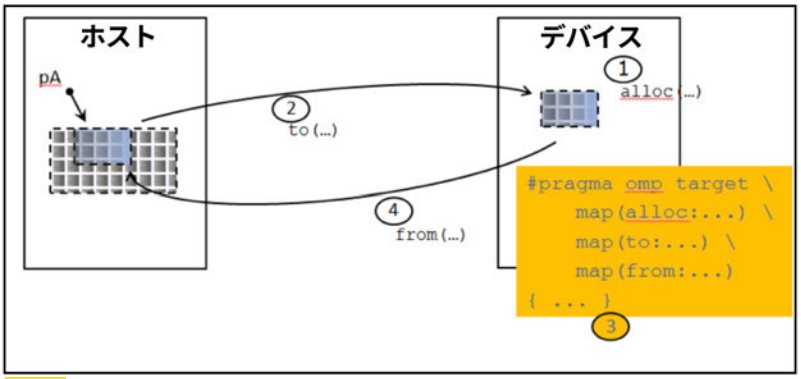
4 伝統的な相互排他 (左) と楽観的な相互排他 (右)

図 3 のコードでは、プログラマーはスレッドがハッシュマップにアクセスする際に競合しないと想定し、ロックを投機的に実行することを OpenMP* ランタイムに伝えるため新しい機能を使用しています。インテル® TSX が利用可能なプロセッサーでは、ハードウェアはロックを取得/解放する代わりに楽観的にロックを無視します (**図 4** を参照)。ハードウェアがロックのセマンティクスの違反を検出した場合にのみ (例えば、あるスレッドがハッシュバケットを変更している間に、ほかのスレッドがそれを読み取った場合)、プロセッサーは遡って排他制御のため伝統的なロックによるコードを再実行します。

投機的なロックを使用する目安として、2 つの基本的なケースが考えられます。1 つは、ロックが競合せず (ハッシュテーブルの例のように) 多くの衝突が発生しない場合に、インテル® TSX と投機的なロックを使用するケースです。2 つ目は、読み取りが多く、書き込みのロック競合がわずかであり、競合が発生しても衝突がほとんどない場合に、投機的なロックによりロックのオーバーヘッドが軽減できるケースです。

オフロード

OpenMP* 4.0 では、コプロセッサーなどの接続された計算デバイスへのオフロードにより、OpenMP* によるマルチスレッド化の範囲が拡がりました。**`target`** 構文は、制御フローをホストスレッドからコプロセッサーへ切り替えます。プログラマーは、**`map`** 節を使用して転送するデータ・オブジェクトと、転送の方向を指定できます (**図 5** を参照)。通常、オフロードされるコード領域はカーネルと呼ばれ、ターゲットデバイスの特性を活用する大規模な並列処理の断片です。



5 データ転送を含むオフロードされたコードの実行モデル

OpenMP* 4.0 では、ホストとターゲット間で無駄なデータ転送を避けるため、異なるオフロードでデータを保持できるデバイスデータ環境を導入しました。**図 6** は、**kernel1** と **kernel2** の呼び出し中に、ターゲットデバイス上の配列 **var1** を継続して利用するデータ領域の例を示しています。データ領域の存続期間は、**target data** 構文に関連付けられたスコープの構造化ブロックに依存します。コードが波括弧 "{" に到達すると、データ環境が作成されデータが転送されます。波括弧 "}" に到達すると、データ環境は破棄されます。これは、複数のカーネルに渡ってデータ環境を維持するための実装を可能にしますが、新たなデータのマッピングや非構造化データのマッピング (例えば、C++ クラスのコンストラクターやデストラクターでの) を禁止します。

```
double var1[N];

void offload_example() {
#pragma omp target data map(tofrom:var1[:N])
    {

        C *c = new C();

#pragma omp target
        c.kernel1();    // var1 と c のメンバーを使用する

#pragma omp target
        c.kernel2();    // var1 と c のメンバーを使用する

        delete c;
    }
}
```

6 **target data** 構文のスコープにバインドされたデバイスデータ環境

OpenMP* 4.5 では、この制限を解除するため新たな構文が追加されました。target enter data と target exit data ディレクティブは、ターゲットデバイス上のデータマッピングを作成および廃棄します。

```
#pragma omp target enter data map(map-type: var-list) [clauses]
#pragma omp target exit data map(map-type: var-list) [clauses]
```

図 7 は、新しいディレクティブを使用して、C++ オブジェクトのコンストラクターとデストラクターでデータ環境を作成および破棄する方法を示しています。**C** クラスのインスタンスが構築されると、メンバーの値がターゲットデータ環境にマッピングされます。オブジェクトが破棄されると、デストラクターは値に関連付けられたデータ環境を破棄します。

その他の拡張は、構造体データ型の要素のマッピングに関するものです。OpenMP* 4.0 では、スカラー変数、配列、またはビット単位でコピー可能なデータ構造体のマッピングのみ許可していました。OpenMP* 4.5 では、構造体データ型のメンバーの部分的なマッピングに対応するため、データ転送が拡張されました。**図 8** にいくつかの新しい可能性を示します。

ついに、OpenMP* 4.5 では非同期オフロードがサポートされました。OpenMP* 4.0 の実装では、ホストスレッドは処理を継続する前に、オフロードされたコード領域の実行完了を待機していました。OpenMP* 4.5 では、**nowait** 節を指定して、ホストのスレッドと **target** 構文の OpenMP* タスクを同時に実行することができます。**target enter data** と **target exit data** ディレクティブも同様に、非同期データ転送を可能にする新しい節をサポートします。

非同期オフロードとデータ転送は、通常の OpenMP* タスクであるため、この新しい機能はホスト上で実行されているほかの OpenMP* タスクと非同期実行を同期する **depend** 節を継承します。**図 9** では、ホストの実行と非同期データ転送がオーバーラップしています。**depend** 節は、データ転送が完了するまでカーネルの実行を待機させます。

```
class C {

 public:
    C() {
#pragma omp target enter data map(alloc:values[M])
    }

    ~C() {
#pragma omp target exit data map(delete:values[M])
    }
 private:
    double *values;
};
```

7 C++ オブジェクトにおけるデバイスデータ環境の作成と破棄

```
struct A {
   int field;
   double array [N];
} a;

#pragma omp target map(a.field)
#pragma omp target map(a.array[23:42])
```

8 構造体データ型の要素のマッピング

```
double data[N];

void synchronization_example() {
#pragma omp target enter data map(to:data[N]) \
            depend(out:data[0]) nowait

    do_something_on_the_host_1();

#pragma omp target depend(inout:data[0]) nowait
    perform_kernel_on_device();

    do_something_on_the_host_2();

#pragma omp target exit data map(from:data[N]) \
            depend(inout:data[0])

#pragma omp task depend(in:data[0])
    task_on_the_host(data);

    do_something_on_the_host_3();
}
```

9 非同期オフロードとデータ転送、ホストスレッドとの同期

また、カーネルの実行が完了する前に、デバイスからホストにデータ転送が行われないようにします。最後に、ホストスレッドは、`do_something_on_the_host_3()` でホストコードを同時実行する前に、最後のデータ転送完了を待機するタスクを生成します。

まとめ

OpenMP* 4.5 は、いくつかの改良に加えて、プログラマーがより良い並列処理を表現し、ホスト・アプリケーションとオフロードコードの両方のパフォーマンス向上を可能にする機能を提供します。タスクを生成するループは、並列ループに起因する負荷バランスと構成の容易性の問題を解決し、プログラマーの負担を軽減します。ヒント付きのロックのサポートは、プログラマーがアプリケーションのロックの動作を最適化し、ハードウェア・トランザクショナル・メモリーに対応した現代のプロセッサーを活用できるようにします。最後に、オフロードの拡張機能は、ホストとコプロセッサーの両方で、計算と通信をオーバーラップする非同期実行を可能にします。

OpenMP* 互換のインテル® C++ / Fortran コンパイラーを評価する

インテル® Parallel Studio XE とインテル® System Studio に含まれます >

インテル® MPI ライブラリー: Hadoop* エコシステムをサポート

MPI で計算負荷の高いタスクを Hadoop* の MapReduce よりも高速に処理

Mikhail Smorkalov 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

長年にわたり **MPI** は、分散処理モデルの主流として利用されてきました。しかし、大量の入力データの処理を必要とするワークロードを扱うハイパフォーマンス・コンピューティング (HPC) により、新しいアプローチとフレームワークが登場しました。最も一般的なものは、Apache* Hadoop* MapReduce[1] であり、特に Hadoop* ソフトウェア・スタック (すべての関連ツールとフレームワークを含む) が挙げられます。

Vanilla Hadoop* は複数のモジュールで構成され、[2] 効率良く解くことができる問題の範囲には一定の制約があります。例えば、MapReduce は、反復間で中間結果をストレージにダンプする必要があるため、反復アルゴリズムには適していません。この欠点を改善するため、反復間で効率良くインメモリー・データをキャッシュできる Apache Spark* や Apache Storm* のような、ほかの Hadoop* フレームワークが開発されました。さらに、クラスター管理フレームワークの YARN* は、MapReduce 以外にも対応しているため、これまで MPI が主流であった分野でも Hadoop* を利用することができます。

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

タスクによっては、Hadoop* スタックを利用することでより簡単に効率良く処理することができ、計算負荷が高く、ノード間通信が多い機械学習でも Hadoop* プラットフォームの採用が増えつつあります。ただし、複数の研究において、[3、4] Hadoop* フレームワークよりも MPI のほうが高速であることが示されています。MPI は HPC 分野にしか適していないと誤解されている方がいらっしゃるかもしれません。この記事は、Hadoop* エコシステムで一般的に使用されているツールを補完するために MPI を利用する利点と課題を明らかにします。

新しい HPC の課題

歴史的に、高性能計算は気象モデル、物理、化学のような、比較的少量の入力を基に大量の計算を行う (場合によっては、大量のデータを出力する)、計算負荷の高い分野で利用されてきました。MPI はそのような問題に最適で、さまざまな通信パターンに加えて、多くの場合、ハードウェア機能を効率良く利用する MPI 実装も提供します。

しかし、新しい分野では、テラバイトやペタバイトのデータを入力し、大量のデータを扱うアプリケーションの開発が求められています。通常、ノード間通信をあまり必要としないため、効率良く、信頼性の高い、スケーラブルな入力/出力 (I/O) が特に重要になります。Hadoop* では、HDFS* によりこれに対応しています。

データ解析タスクには長い時間がかかるため、透過的なフォールト管理が必須です。さらに、ストリーミング・タスクは永続的に実行可能なため、ノードクラッシュは避けられません。Hadoop* ソフトウェア・スタックは、ジョブ全体を再起動しなくても、タスクを円滑に引き渡すことができるメカニズムを採用しています。失敗したサブタスクのみ正常なノードに渡されます。また、Hadoop* スケジューラーは、タスクをできるだけデータの近くで実行するため、最適なデータの局所性が得られます。

これらすべての機能が、データ・サイエンティストにとって Hadoop* プラットフォームを魅力的なものにし、Hadoop* エコシステムの解析フレームワークとライブラリーの開発を促進しました。フレームワークは、さまざまなプログラミング言語 (Python*、Java*、Scala* など) に対応しているため、簡単に導入することができます。従来の MPI 実装は、MPI 標準で定められている C と Fortran のみに対応していますが、Java* や Python* インターフェイスを提供する MPI 実装もあります。

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

MPI レパートリー

最初に、MapReduce プログラムの実装に利用できる、豊富な MPI 通信パターンについて考えてみましょう。Map フェーズは MPI_Scatter(v) 関数を利用して実装できます。Reduce フェーズには MPI_Reduce[5] か、リダクションを実行する前にすべてのデータが揃っていなければならない場合は MPI_Alltoall(v) とマージ操作を利用します。MPI 実装は通常、集合操作ごとにいくつかのアルゴリズムがあり、スケール、メッセージサイズ、ハードウェア・アーキテクチャーに応じて最適なものを選択します。そのため、Hadoop* で使用される主な操作は MPI とよく似ています。ビッグデータに不可欠なフォールト管理とデータ管理の機能はどうでしょうか? MPI にそれらに相当する機能はあるのでしょうか?

Hadoop* は、データの局所性、つまり、できるだけプロセスに近いストレージからデータを取得することによって優れたスケーラビリティーを達成します。一方、HPC は、ネットワーク経由でアクセスする並列ファイルシステムを利用するため、大量の入力データを扱う場合、ネットワークの帯域幅とロードバランスが非常に重要になります。ただし、最近の並列ファイルシステム (Lustre* や GPFS* など) は、システムサイズの増加に伴い、ほぼ線形のスケーラビリティーを提供すると言われています。つまり、スケーラブルなデータ管理に利用できます。インテルが提供する Lustre* は、Lustre* ファイルシステムを利用する Hadoop* で最適なデータアクセスを実現します。[6] Hadoop* アプリケーションの実行で、常に完全なデータの局所性を保持することは不可能ですが、並列ファイルシステムはリモートデータへの高速な並列アクセスを可能にします。

Hadoop* でも HDFS* 以外のファイルシステムを利用しているケースがあります。[7、8] HPC クラスターで利用されているストレージシステムの並列性と高帯域幅インターコネクトは、Hadoop* にも利点をもたらします。例えば、[8] Lustre FS* は、コモディティー・ノード上のローカル・ドライブ・アクセスよりも高速なデータアクセスを提供します。

さらに、MPI-IO には、柔軟にチューニング可能な集合 I/O 操作が含まれています。例えば、集合バッファリングにより、ノード上の 1 つのプロセスのみが FS への書き込み/読み取りを行うようにし、I/O の同時実行を減らすことができます。

フォールトトレラントは、Hadoop* プラットフォームの最大の利点の 1 つと言え、ハードウェア障害からほぼ透過的に回復することができます。同時に、フォールトトレラントは MPI 標準の弱点でもあります。実装によっては、特殊なワークフローに従ってフォールトトレラントの MPI プログラムを記述することも可能でしょうが、[9] そのためにはアプリケーション・エンジニアによる多大な労力を必要とします。MPI 4.0 で導入される可能性があるユーザーレベルのフォールトトレラントは、透過的なフォールト管理を提供しませんが、少なくとも MPI 開発者の負担を軽減できるでしょう。MPI で利用可能な唯一の透過的なフォールト管理メカニズムは、チェックポイントです。[10] グローバル・スナップショットに基づいて、すべてのタスク (正常なものも含む) がチェックポイントから再起動されます。これは、Hadoop* と比較した MPI 実装の数少ない欠点の 1 つです。ストリーミング処理アプリケーションのような実行時間の長いサービスには MPI は適していません。

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

Hadoop* エコシステムのもう 1 つの利点は、さまざまな分野にわたる広範なコミュニティーの存在です。ファイル、リレーショナル・データベース、ストリームなど、さまざまなデータソースへの簡単なアクセスを提供する多数のツールとテクノロジーを利用できます。一方、MPI は、パフォーマンス重視の比較的低水準のテクノロジーであるため、API で多種多様な構文は用意されていません。ただし、インテル® Data Analytics Acceleration Library (インテル® DAAL)[11] のようなテクノロジーにより、さまざまなデータソースからデータを読み取り、変換し、処理する、データフロー全体を効率良く実装する方法 (広範なアルゴリズムを含む) が提供されています。これは、MPI によりパフォーマンスを向上したいと思いながらも、すべてのデータフローを自身で実装しなければならないため諦めていた開発者にとっては朗報と言えます。

また、Vanilla Hadoop* は、シャッフルフェーズでノード間通信に TCP/IP を利用するため、高速インターコネクトの利点を得ることは容易ではありません。ただし、ノード間通信に RDMA を利用することでこの問題を緩和できます。[12]

多くの方は、タスク処理ツールを選択する際に、スケーラビリティーに優れた Hadoop* を検討するかもしれませんが、MPI も適していると言えます。例えば、**インテル® MPI ライブラリー**は、最大 340,000 プロセスまでスケーリングします。[13]

Hadoop* エコシステムで MPI を実行する

前述のように、MPI は新しい HPC の課題に対応することが可能で、特定の分野では Apache* Spark* の代わりに利用できます。しかし、HPC と Hadoop* は、それぞれの分野で一般的なアプリケーションのニーズに応じて並行して進化してきたため、リソース・マネージャーを含め異なるエコシステムを利用しており、MPI と Hadoop* のツールやフレームワークを組み合わせて使用することは容易ではありません。HPC と Hadoop* を融合させるため現在行われている取り組みは、主に Hadoop* のツールを HPC 環境に移行し (または、MR-MPI[14] のように MPI で新たに MapReduce フレームワークを実装し)、ハイエンドシステムによってもたらされるパフォーマンスの利点を利用しようとするものです。データ解析や機械学習アプリケーションを HPC クラスター上で実行して処理を高速化することは 1 つの選択肢ですが (そして、これは最も確実であると考えられますが)、[15] すでに Hadoop* インフラストラクチャーを導入し、運用に成功している企業にとってその利点はないでしょう。

別の選択肢は、クラスター上に異なる 2 つのインフラストラクチャーを保持せずに MPI アプリケーションを Hadoop* 環境で実行するか、2 つの異なるクラスターを構築することです。[16] このセクションでは、MPI を Cloudera Distribution Including Apache Hadoop* (CDH) 上の Hadoop* エコシステムに統合するためのメカニズムを説明します。[17]

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

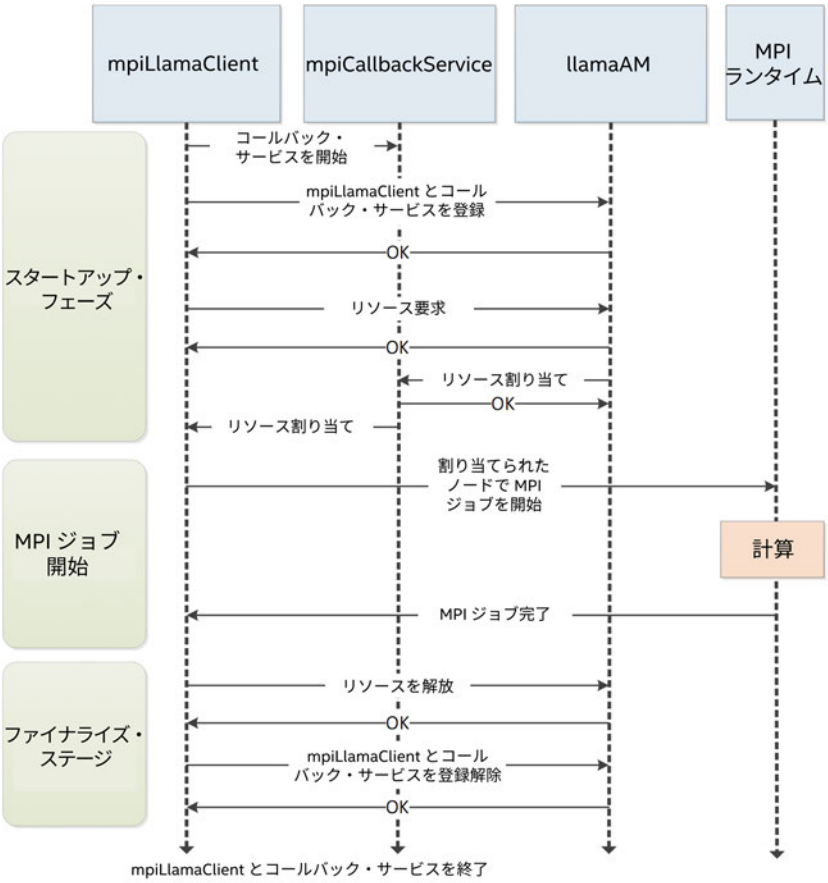
CDH パッケージには、Llama* が含まれています。[18] これは、YARN* のアプリケーション・マスターで、元々 Impala* 用に設計されたものです。[19] Llama* は、リソースの要求/解放とその他の機能 (MPI の実行に不可欠なギャング・スケジュールなど) に関連したクロス言語の Apache* Thrift* API を提供します。この機能セットにより、プログラムで Hadoop* クラスターノードに関する情報を取得し、必要に応じて MPI ジョブ用のリソースを要求して、MPI と Hadoop* ジョブが同一のインフラストラクチャーを動的に共有できるようにします。

Llama* の機能を利用して Hadoop* クラスター上で MPI を実行するには、2 つの補完サービスを実装する必要があります。

- **MPI Llama* クライアント:** 必要な情報 (クラスターノード名など) の取得とリソース要求/解放を Llama* に照会するエンティティー。
- **MPI Llama* コールバック・サービス:** 特定のイベント (ノード割り当てなど) に関する Llama* からの通知を待機するデーモン。

実際のワークフローは、3 つの独立したフェーズで構成されます (**図 1**)。

- **スタートアップ:** 補完サービス (クライアントとコールバック) を起動し、Llama* に登録して、MPI ジョブに必要なリソースを要求します。このフェーズは、コールバック・サービスがリソースの割り当て完了通知を受け取ると終了します。
- **MPI ジョブの開始:** Llama* によって提供されたリソースリストを基に、Hadoop* クラスターノード上で MPI ジョブをネイティブ実行します。
- **ファイナライズ:** MPI ジョブが完了したらリソースを解放し、補完サービスを終了します。



1 Hadoop* クラスターでの MPI の実行

このメカニズムの利点の 1 つは、Llama* の通信時間は MPI アプリケーションの複雑さに依存しないため、純粋な MPI 通信と比べてオーバーヘッドが一定であることです。実際のアプリケーションでは、合計ウォールクロック時間に占める Llama* の通信オーバーヘッドの割合はわずかであるため、MPI と Hadoop*/Spark* アプリケーションのパフォーマンス比較に関連するすべての結果に影響しません。

ほかのアプローチ (mpich2-yarn[20] プロジェクトなど) と比較したこのアプローチのもう 1 つの利点は、リソース要求と MPI ジョブの開始をワークフローの別々のフェーズで行っているため、特定の MPI 実装に縛られないことです。つまり、MPI の起動コマンドを変更するだけで、MPI 実装を切り替えることができます。例えば、MPI 実装として BDMPI[21] を利用し、MPI でアウトオブコア・アルゴリズムを効率良く実行して、ネイティブ Hadoop* フレームワークの代わりに、ビッグデータ問題に柔軟に対応できます。

前述の機能は、インテル® MPI ライブラリー 5.1 Update 2 で実装されています (使用モデルについては、『**インテル® MPI ライブラリー・リファレンス・ガイド**』を参照してください)。

Hadoop* クラスター上で MPI を実行する場合、一般にいくつかの制限があります。

- MPI 実装は HDFS をサポートしていないため、MPI-IO は共有フォルダーでのみ、またはローカルで利用できます。そのため、クラスターの管理者は、HDFS に加えて、NFS フォルダーもセットアップしたほうが良いでしょう。
- YARN* は、コンテナーの CPU アフィニティー情報を提供しないため、同じノードで複数のアプリケーションを実行する場合、ピニング機能の使用には細心の注意を払うべきです。

パフォーマンスの評価

データ解析でよく使用される 2 つの分散処理アルゴリズムを使用します。

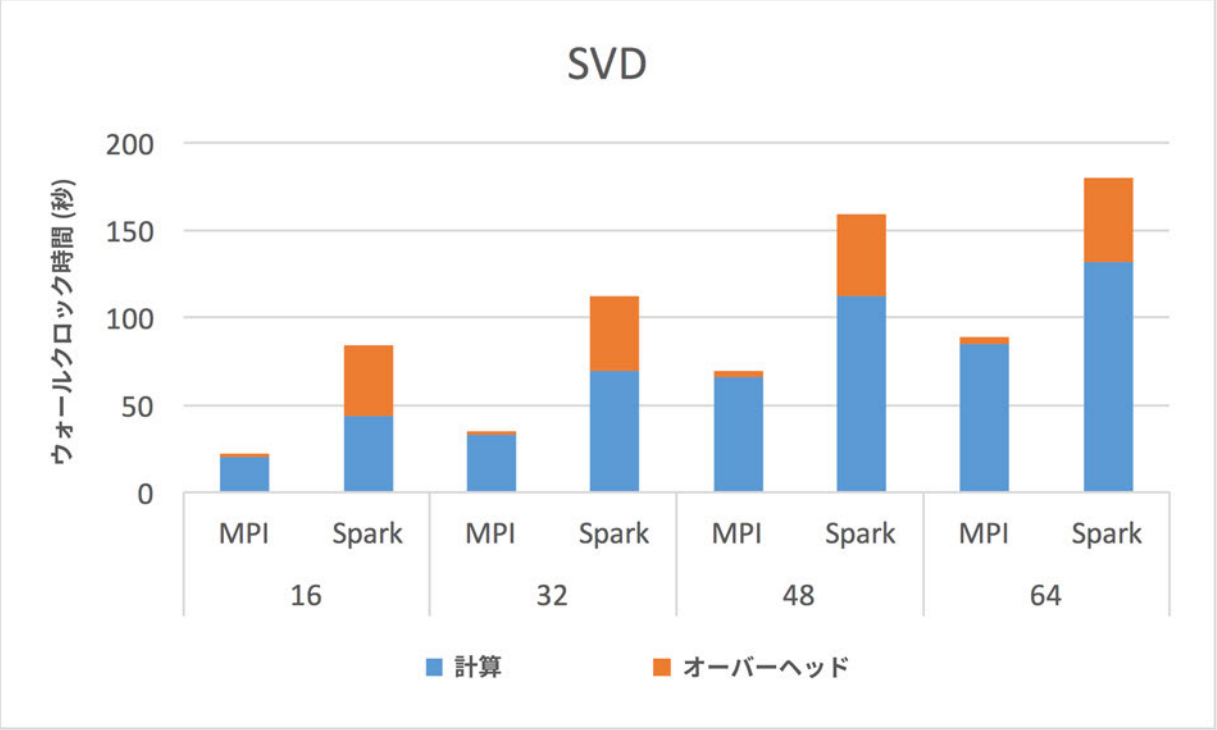
1. 特異値分解 (SVD)
2. 主成分分析 (PCA、相関メソッドを使用)

インテル® DAAL に含まれる各アルゴリズムの Spark* および MPI 向けのサンプルを使用します。これらのサンプルは、同じビルディング・ブロックで構成されているため、実装の詳細ではなく、フレームワーク (Spark* と MPI) および言語 (Java* と C) によるパフォーマンスの違いを説明することができます。Spark* 用の SVD サンプルは、ドライバーノードで左直交行列を収集しないように変更しました。この収集は一般に過剰にデータを収集し、Spark* RDD の利点が得られないためです (そのため、Spark* 用アルゴリズムは、MPI 用アルゴリズムよりも複雑さが緩和されています)。

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

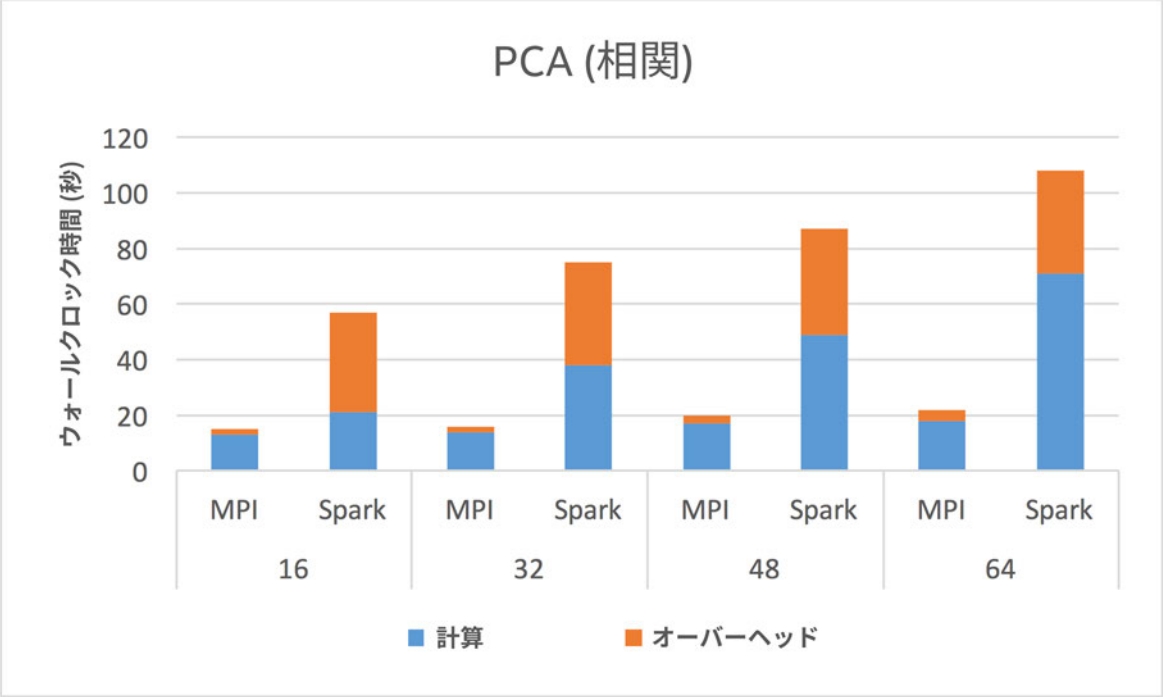
パフォーマンスの評価には、8 ノードのクラスターを使用しました。各ノードには、1 基のインテル® Xeon® プロセッサー X5570、1 つの NetEffect* NE020 10GB イーサネット・アダプター、12GB RAM が搭載されています。また、動作環境として、SLES* 11.0 Linux* オペレーティング・システム、CDH の Cloudera Express* 5.4.6 バージョン (Spark* 1.3.0)、インテル® MPI ライブラリー 5.1.2、インテル® DAAL 2016 Update 1、インテル® C++ コンパイラー 15.0.4、JDK 1.7.0_67、Scala* 2.10.4 を使用しました。

それぞれ 10,000 x 1,000 要素の入力行列を含む、16、32、48、および 64 データブロックでパフォーマンスを測定しました。64 ブロックのデータセットのサイズは 4.2GB です。MPI サンプルの実行には、1 ブロックにつき 1 MPI プロセスを開始しました。Spark* では、spark.executor.instances プロパティーでエグゼキューターの数をブロック数に設定しました。Spark* は独自のヒューリスティックに従ってエグゼキューターの数を定義するため、常にこの設定が尊重されるわけではありません。また、Spark* は JVM と独自の設定に影響されやすいため、経験に基づいて、パフォーマンスを最適化するようにメモリー設定を選択しました (**図 2** と **3** を参照)。



2 特異値分解

spark.executor.memory = 1300m
spark.yarn.executor.memoryOverhead = 1024m
spark.kryoserializer.buffer.max.mb = 512m



3 主成分分析

spark.executor.memory = 1024m
spark.yarn.executor.memoryOverhead = 768m

上記の合計ウォールクロック時間は、実際の計算処理とオーバーヘッドから成ります。MPI サンプルのオーバーヘッドは、インテル® MPI ライブラリーが Llama* とリソースについてネゴシエートし、MPI ジョブを開始するのに必要な時間で、Spark* サンプルのオーバーヘッドは、ウォールクロック時間からすべての計算処理ステージの実行時間の合計を引いたものです。

MPI は、オーバーサブスクリプション・モードでの実行の影響を受けやすいため、MPI ランクの数が物理コア数よりも大きい場合、パフォーマンスがやや低下することがあります。それでも、ウォールクロック時間が Spark* よりも短いことが分かります。

まとめ

パフォーマンスの評価結果と参考文献の結果は、[3、4] MPI がデータ解析で使用される一部のアルゴリズムに適していることを示しています。また、この記事で紹介したように MPI を Hadoop* エコシステムに統合することで、この分野での MPI の利用を拡大できます。

ここで重要なことは、MPI を Hadoop* ベースのフレームワークの代わりに使用し、優れたパフォーマンスが得られる問題もありますが、そうでない問題もあるということです。そのため、2 つを組み合わせて利用することは自然であり、それにより新たな相乗効果が得られます。実行時間の長いサービスは Hadoop* ツールで実装し、計算負荷が高く、複雑な通信を含む、比較的実行時間の短いタスクには MPI を利用します。

MPI は、優れたパフォーマンスをもたらすだけでなく、Hadoop* 開発者は、長年にわたって記述された数百万行の MPI コードを活用することができます。さらに、データ解析が HPC 分野に拡大し、新たなハイパフォーマンス・データ解析分野が出現しつつある中、MPI とビッグデータ・フレームワークを同じエコシステムで利用すべきかどうかという問題は、もっと注目されるべきです。MPI と Hadoop* エコシステムの統合は、その 1 つの選択肢と言えます。

BLOG HIGHLIGHTS

コードの現代化を成功させるための 3 つのアドバイス

CLAY BRESHEARS >

プログラムを高速化するための開発者への 3 つのアドバイスについてブログの執筆依頼を受け取ったとき、最初に思い浮かんだのは「Location! Location! Location!」という言葉でした。そこから不動産を連想し、映画「Glengarry Glen Ross (邦題: 摩天楼を夢見て)」のブレイク (Alec Baldwin) の台詞「A-B-C. A-Always, B-Be, C-Concurrent. Always be concurrent.」を思い出しました。

この単純な台詞には、さまざまなものが凝縮されています。私はこれまで、粒度、ロードバランス、タスク分割、ループの並列化など、個々のトピックについて多数の IDZ ブログを執筆してきましたが、正にそれらがアドバイスと言えます。(それらのブログを見つけ、そのうち 3 つを読めば、この記事を最後まで読む必要はないかもしれません。あるいは、この記事を読んだ後に、個別のブログで詳細を確認するほうが良いでしょう。) また、コンカレントおよび並列プログラミングに関する本も執筆しました。お手元にある方は、後半の任意の 3 つのページをお読みになれば、コードを高速化するための 3 つのヒントが得られるでしょう。

この記事の続きはこちらでご覧になれます。 >

参考文献

1. J. Dean and S. Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters.” Comm. ACM, 51(1): 107–113, January 2008.
2. Apache Hadoop Wiki, **hadoop.apache.org/#What+Is+Apache+Hadoop%3F**.
3. S. Jha, J. Qiu, A. Luckow, P. Mantha, and G. C. Fox. “A Tale of Two Data-Intensive Paradigms: Applications, Abstractions, and Architectures,” eprint arXiv:1403.1528, March 2014.
4. F. Liang, C. Feng, X. Lu, and Z. Xu. “Performance Benefits of DataMPI: A Case Study with BigDataBench.” In The 4th Workshop on Big Data Benchmarks, Performance Optimization, and Emerging Hardware, BPOE-4, Salt Lake City, Utah, 2014.
5. T. Hoefler, A. Lumsdaine, and J. Dongarra. “Towards Efficient MapReduce Using MPI.” Recent Advances in Parallel Virtual Machine and Message Passing Interface, pp. 240–249, 2009.
6. “Intel® Enterprise Edition for Lustre*,” **intel.com/content/www/us/en/software/intel-enterprise-edition-for-lustre-software.html**.
7. A. Woodie. “What Can GPFS on Hadoop Do For You?” Datanami, February 2014, **datanami.com/2014/02/18/what_can_gpfs_on_hadoop_do_for_you_/**.
8. N. Rutman. “Map/Reduce on Lustre: Hadoop Performance in HPC Environments” (technical white paper). Xyratex, **xyratex.com/sites/default/files/Xyratex_white_paper_MapReduce_1-4.pdf**.
9. W. Gropp and E. Lusk. “Fault Tolerance in MPI Programs.” The International Journal of High Performance Computing Applications, Volume 18, No. 3, Fall 2004, pp. 363–372.
10. J. Hursey, J. M. Squyres, and A. Lumsdaine. “A Checkpoint and Restart Service Specification for Open MPI” (technical report), **open-mpi.org/papers/iu-cs-tr635/iu-cs-tr635.pdf**.
11. ベータ版 インテル® Data Analytics Acceleration Library 2016 のリリース, **isus.jp/products/daal/announcing-intel-daal-2016-beta**.
12. High-Performance Big Data Project, Network-Based Computing Laboratory, Ohio State University. “RDMA-Based Apache Hadoop,” **hibd.cse.ohio-state.edu**.
13. インテル® MPI ライブラリー, **isus.jp/intel-mpi-library**.
14. S. Plimpton. “MapReduce and MPI.” SOS 17 - Intersection of HPC & Big Data, March 2013.
15. J. Dursi. “HPC Is Dying, and MPI Is Killing It.” **dursi.ca/hpc-is-dying-and-mpi-is-killing-it**.
16. The Nielsen Company. “Bridging the Worlds of High Performance Computing and Big Data,” **sites.nielsen.com/newscenter/bridging-the-worlds-of-high-performance-computing-and-big-data**.
17. “CDH Components,” **cloudera.com/content/cloudera/en/products-and-services/cdh.html**.
18. Cloudera, Inc. “Llama,” **cloudera.github.io/llama**.
19. Cloudera, Inc. “Apache Impala,” **impala.io**.
20. GitHub, Inc. “mpich2-yarn,” **github.com/alibaba/mpich2-yarn**.
21. Karypis Lab. “BDMPI - Big Data Message Passing Interface,” **glaros.dtc.umn.edu/gkhome/bdmpi/overview**.

インテル® MPI ライブラリーを評価する

インテル® Parallel Studio XE Cluster Edition に含まれます >

メモリーアクセスのパフォーマンス・ボトルネックの検出

インテル® VTune™ Amplifier XE の新しいメモリーアクセス解析により アプリケーションのパフォーマンスを素早く簡単に向上

Kevin O'Leary 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
Dmitry Ryabtsev 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
Alexey Budankov 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

アプリケーションのメモリーアクセスは、パフォーマンスに大きく影響します。スレッド化と**ベクトル化**によりアプリケーションを並列化するだけでは十分ではありません。メモリー帯域幅も重要ですが、多くの場合、ソフトウェア開発者はそのことをよく理解していません。メモリー・レイテンシーを最小限に抑え、帯域幅の向上を支援するツールを利用することで、パフォーマンス・ボトルネックをピンポイントで特定し、その原因を診断することができます。

最近のプロセッサーは、多様なメモリーアクセスを行います。例えば、L1 キャッシュヒットのレイテンシーは、すべてのキャッシュをミスして DRAM にアクセスしなければならない場合のレイテンシーとは大きく異なります。不均等メモリーアクセス (NUMA) アーキテクチャーではさらに複雑さが増します。

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

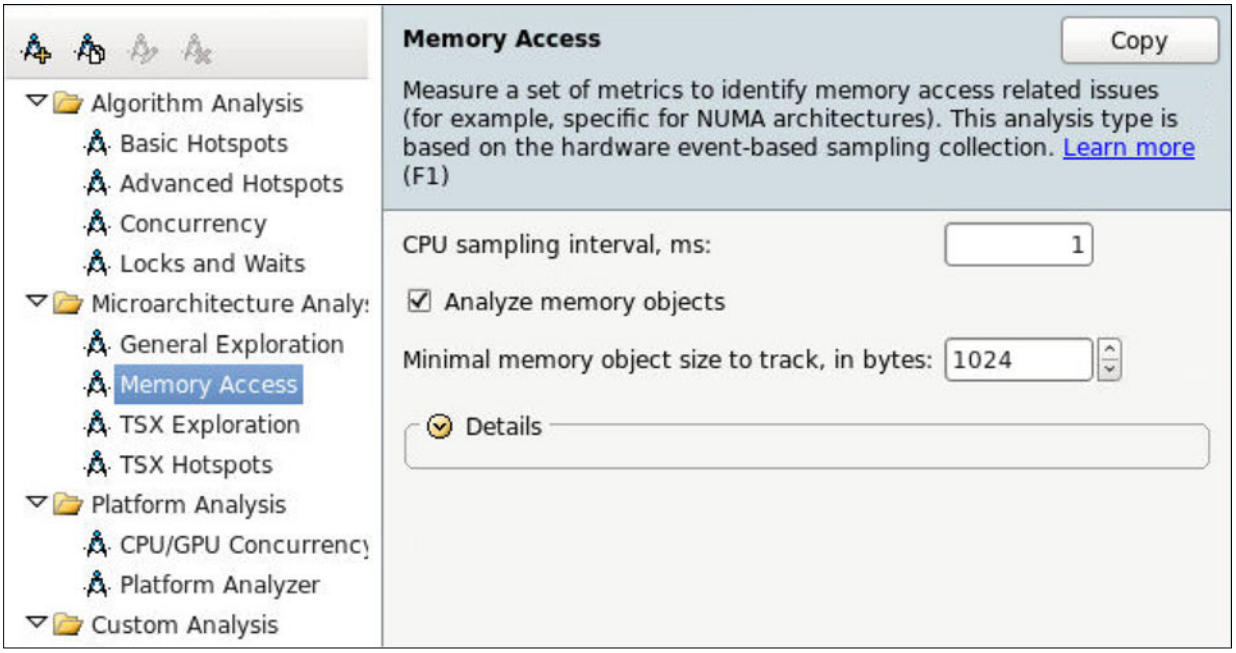
インテル® VTune™ Amplifier XE は、メモリーアクセスの解析を支援するさまざまな機能を備えたパフォーマンス・プロファイラーです。新しいメモリーアクセス解析タイプに含まれる各種機能を利用して、次のことが可能です。

- メモリー階層で**パフォーマンスの問題を検出** (L1-、L2-、LLC-、DRAM-バウンドなど)。
- **メモリー・オブジェクトを追跡**し、そのレイテンシーを適切なコードとデータ構造に関連付けます。
- **帯域幅により制限されたアクセスを解析**し (DRAM とインテル® QuickPath インターコネクト [インテル® QPI] の帯域幅を含む)、帯域幅をプログラムのタイムラインで表示した DRAM およびインテル® QPI のグラフとヒストグラムを素早く確認できます。
- パフォーマンスに影響する **NUMA 関連の問題**を特定できます。

この記事では、新しいメモリーアクセス機能の概要と、この機能によりいくつかの困難なメモリーの問題を解決し、アプリケーションのパフォーマンスを大幅に向上する方法を説明します。

はじめに

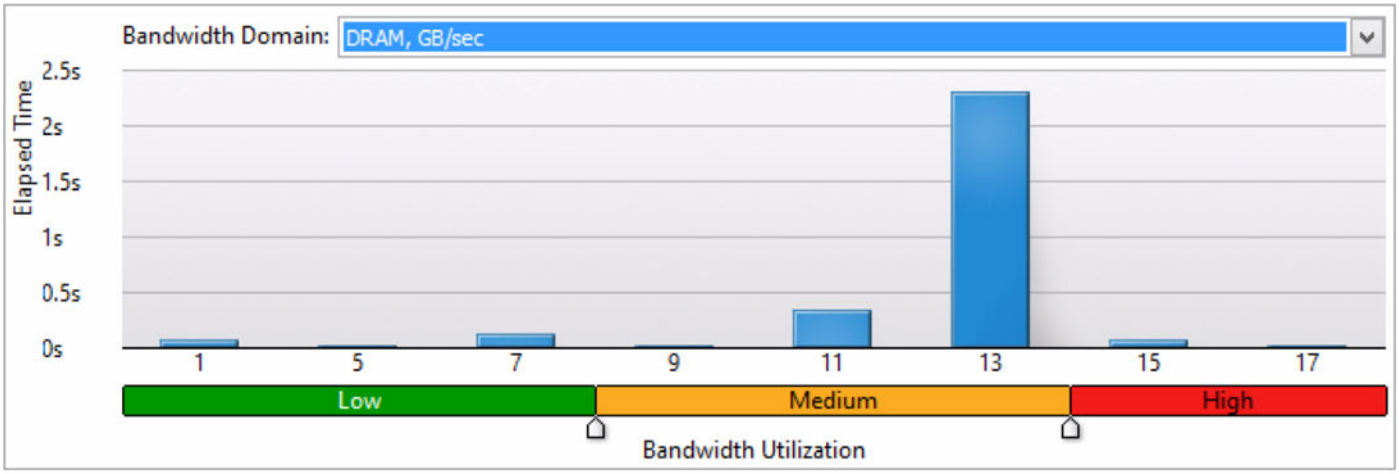
インテル® VTune™ Amplifier XE のメモリーアクセス機能を使用するには、新しい [Memory Access (メモリーアクセス)] 解析タイプをクリックして、[Start (開始)] をクリックします (**図 1**)。



1 メモリーアクセス機能を利用する

帯域幅使用率を表示する

DRAM およびインテル® QPI の帯域幅の使用状況を確認できます。帯域幅の使用率が高い箇所に注目します。この問題を修正するには、帯域幅に影響しているコード領域を見つけます (**図 2**)。



2 帯域幅ヒストグラム

帯域幅に影響するメモリー・オブジェクトを確認する

帯域幅に影響しているソースコードとメモリー・オブジェクトを特定します。帯域幅ドメインでグループ化すると、メモリー帯域幅に最も影響しているメモリー・オブジェクトを特定できます (**図 3**)。DRAM やインテル® QPI 関連の問題を含むコード領域を確認できます。

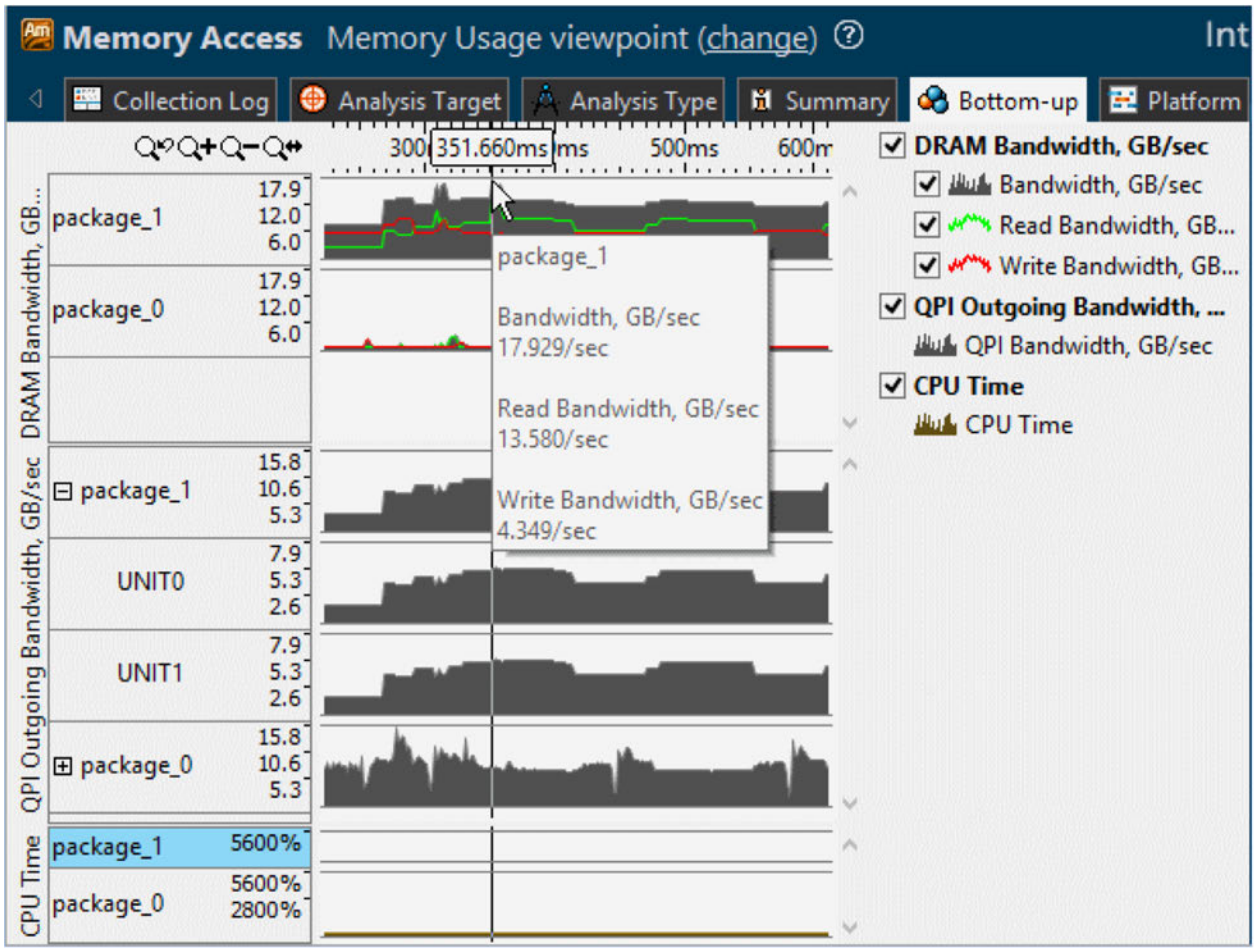
Grouping: Bandwidth Domain / Bandwidth Utilization Type / Memory Object / Allocation Stack

Bandwidth Domain / Bandwidth Utilization Type / Memory Object / Allocation Stack	CPU Time	Memory Bound	Loads	Stores	LLC Miss Count	Average Latency (cycles)
DRAM, GB/sec	2.873s	0.818	1,764,005,292	846,012,690	37,202,232	195
High	2.291s	0.924	1,176,003,528	601,209,018	33,602,016	196
lin_stream.cpp:99 (152 MI			300,000,900	140,402,106	12,600,756	316
lin_stream.cpp:100 (152 M			426,001,278	261,603,924	12,000,720	199
lin_stream.cpp:98 (152 MI			444,001,332	193,202,898	9,000,540	73

3 メモリー帯域幅

メモリー帯域幅をアプリケーションのタイムラインで表示する

メモリー帯域幅は、一般にプログラムの実行とともに変動します。読み取り/書き込みの帯域幅を GB/秒で表示したグラフ (**図 4**) から、アプリケーションのメモリー使用量が増加した場所を確認し、その領域に注目します。タイムラインでメモリー使用量が増加した場所を選択してフィルターし、その時間に動作していたコードを確認します。



4 メモリー使用状況

メモリー帯域幅に影響しているアプリケーションのコード領域を追跡する機能は非常に役立ちます。平均レイテンシーはメモリーアクセスをチューニングする際の重要なメトリックです。タイムライン・グラフで帯域幅を表示すると、アプリケーション実行時のメモリー使用量を簡単に区別することができます。インテル® VTune™ Amplifier XE の最新バージョンでは、帯域幅グラフがプラットフォームで可能な最大値に対して相対的に表示されるため、パフォーマンス向上の可能性を簡単に把握できます。

メモリーの問題を解決する

困難な問題 1: フォルス・シェアリング

最初に、用語を説明しましょう。

シェアリング (共有): 複数のスレッドが同じメモリー位置にアクセスする場合、それらのスレッドはメモリーを「シェア」していると言います。最近のコンピューターの構造上、シェアリングはさまざまなパフォーマンス・ペナルティーを引き起こします。すべてのスレッド/コアがメモリーアドレスに格納されるものについてコミットし、競合に関連するすべてのキャッシュを同期しなければならないため、これらのパフォーマンス・ペナルティーが生じます。

フォルス・シェアリング: 2 つのスレッドが同じキャッシュライン上のメモリー位置にアクセスする場合に発生します。実際に同じメモリー位置をシェアしているわけではありませんが、メモリー参照が互いに近い位置にあるため、同じキャッシュライン上に格納されています。複数のスレッドでフォルス・シェアリングが発生すると、実際に同じメモリー位置をシェアしている場合と同様のパフォーマンス・ペナルティーが生じます (このパフォーマンス・ペナルティーは完全に不要なものと言えます)。

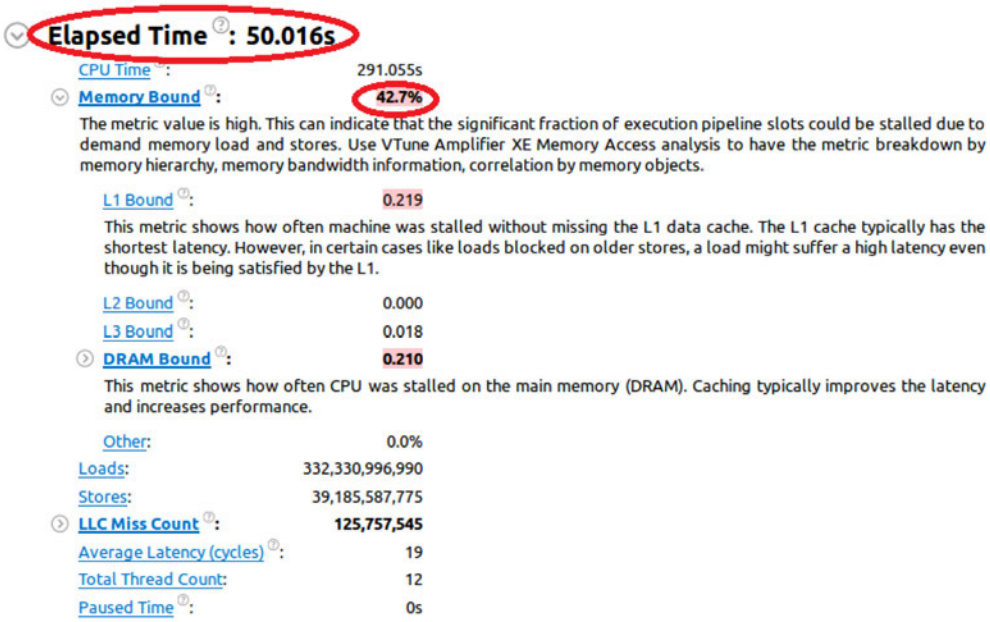
ここでは、Phoenix System* の linear_regression アプリケーションを検証します (**csl.stanford.edu/~christos/sw/phoenix**)。

ステップ 1: メモリーアクセス解析を実行して潜在的なメモリーの問題を明らかにする

次の設定で、メモリーアクセス解析を実行します。

- [Memory Objects Analysis (メモリー・オブジェクト解析)] を選択します。
- すべてのメモリー割り当てをキャプチャーするように、[Object Size Threshold (オブジェクト・サイズのしきい値)] を 1 に設定します。

[Summary (サマリー)] ビューに主要メトリックが表示されます (**図 5**)。

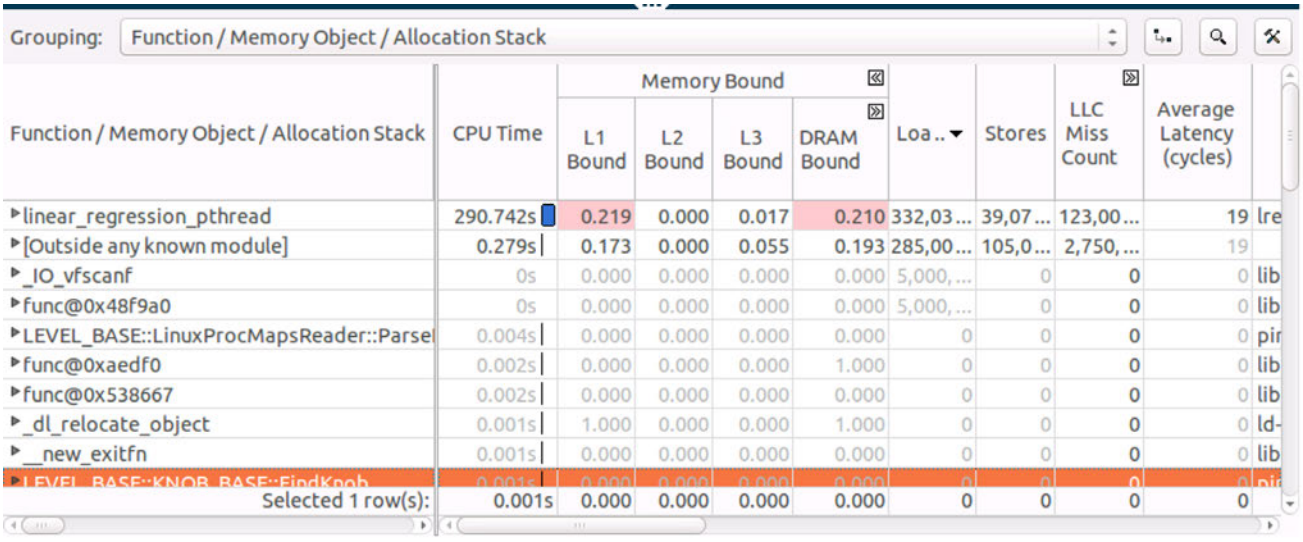


5 主要メトリック

最初の実行の経過時間は 50 秒です。また、アプリケーションが「メモリーバウンド (依存)」であり、メモリー操作の完了を待機するため、CPU の 42% 以上が無駄になっていることが分かります。Memory Bound (メモリーバウンド) メトリックはピンク色で表示されており、対応が必要なパフォーマンスの問題が存在する可能性を示しています。

ステップ 2: 見つかったメモリーの問題を調査する

[Bottom-Up (ボトムアップ)] タブに切り替えて (**図 6**) 詳細を確認します。



Grouping: Function / Memory Object / Allocation Stack

Function / Memory Object / Allocation Stack	CPU Time	Memory Bound: L1 Bound	L2 Bound	L3 Bound	DRAM Bound	Loa..	Stores	LLC Miss Count	Average Latency (cycles)	
linear_regression_pthread	290.742s	0.219	0.000	0.017	0.210	332,03...	39,07...	123,00...	19	lre
[Outside any known module]	0.279s	0.173	0.000	0.055	0.193	285,00...	105,0...	2,750,...	19	
_IO_vfscanf	0s	0.000	0.000	0.000	0.000	5,000,...	0	0	0	lib
func@0x48f9a0	0s	0.000	0.000	0.000	0.000	5,000,...	0	0	0	lib
LEVEL_BASE::LinuxProcMapsReader::Parse	0.004s	0.000	0.000	0.000	0.000	0	0	0	0	pir
func@0xaedf0	0.002s	0.000	0.000	0.000	1.000	0	0	0	0	lib
func@0x538667	0.002s	0.000	0.000	0.000	0.000	0	0	0	0	lib
_dl_relocate_object	0.001s	1.000	0.000	0.000	1.000	0	0	0	0	ld-
__new_exitfn	0.001s	0.000	0.000	0.000	0.000	0	0	0	0	lib
LEVEL_BASE::KNOB_BASE::FindKnob	0.001s	0.000	0.000	0.000	0.000	0	0	0	0	pir
Selected 1 row(s):	0.001s	0.000	0.000	0.000	0.000	0	0	0	0	

6 [Bottom-Up (ボトムアップ)] タブ

実行時間のほとんどが、1 つの関数 linear_regression_pthread で費やされていることが分かります。この関数は、L1 と DRAM に依存しています。

linear_regression_pthread 関数の行を展開すると、アクセスしたメモリー・オブジェクトがロード数順に表示されます (**図 7**)。

Grouping: Function / Memory Object / Allocation Stack

Function / Memory Object / Allocation Stack	CPU Time	Memory Bound					Stores	LLC Miss Count	Average Latency (cycles)	
		L1 Bound	L2 Bound	L3 Bound	DRAM Bound	Loa..				
linear_regression_pthread	290.742s	0.219	0.000	0.017	0.210	332,03...	39,07...	123,00...	19	lre
stddefines.h:52 (512 B)						218,96...	38,29...	250,015	44	lre
[Stack]						87,120...	769,0...	0	8	lre
linear_regression_pthread.c:118 (517 MB)						25,930...	0	121,50...	11	lre
[Unknown]						25,000...	15,00...	1,250,...	0	lre
[Outside any known module]	0.279s	0.173	0.000	0.055	0.193	285,00...	105,0...	2,750,...	19	
_IO_vfscanf	0s	0.000	0.000	0.000	0.000	5,000,...	0	0	0	lib
func@0x48f9a0	0s	0.000	0.000	0.000	0.000	5,000,...	0	0	0	lib
LEVEL_BASE::LinuxProcMapsReader::Parse	0.004s	0.000	0.000	0.000	0.000	0	0	0	0	pir
func@0xaedf0	0.002s	0.000	0.000	0.000	1.000	0	0	0	0	lib
Selected 1 row(s):						218,96...	38,29...	250,015	44	

7 行を展開してロード数順に表示

最も時間がかかっているオブジェクト—stddefines.h:52 (512B)—はたった 512 バイトで、L1 キャッシュに収めることができるはずですが、Average Latency (平均レイテンシー) メトリックは 44 サイクルになっています。これは、通常の L1 アクセス・レイテンシーである 4 サイクルをはるかに超えています。このような状況では、多くの場合、シェアリングまたはフォルス・シェアリングによる競合が発生している可能性があります。

"stddefines.h:52 (512B)" オブジェクトの割り当てスタックを確認すると (**図 8**)、オブジェクトの割り当てが行われたソースの位置が分かります。

```
127     num_threads = num_procs;
128
129     printf("Linear Regression P-Threads: Running...\n");
130
131
132     POINT_T *points = (POINT_T*)fdata;
133     long long n = (long long) finfo.st_size / sizeof(POINT_T);
134
135     req_units = n / num_threads;
136     tid_args = (lreg_args *)CALLOC(sizeof(lreg_args), num_procs);
```

8 ソースの位置

この例では、num_procs はスレッド数で、割り当てられた構造体は lreg_args です。

```
typedef struct
{
    pthread_t tid;
    POINT_T *points;
    int num_elems;
    long long SX;
    long long SY;
    long long SXX;
    long long SYY;
    long long SXY;
} lreg_args;
```

スレッドは、次のように lreg_args 構造体にアクセスしています。

```
 // ADD UP RESULTS
for (i = 0; i < args->num_elems; i++)
{
   //Compute SX, SY, SYY, SXX, SXY
   args->SX  += args->points[i].x;
   args->SXX += args->points[i].x*args->points[i].x;
   args->SY  += args->points[i].y;
   args->SYY += args->points[i].y*args->points[i].y;
   args->SXY += args->points[i].x*args->points[i].y;
}
```

各スレッドは、配列の別々の要素にアクセスしているため、フォルス・シェアリングと考えられます。

ステップ 3: フォルス・シェアリングを排除するようにコードを変更する

フォルス・シェアリングは通常、スレッドが異なるキャッシュラインにアクセスするように、パディングを追加することで簡単に回避できます。

次のように、lreg_args 構造体に **`char pad[80]`** フィールドを追加します。

```
typedef struct
{
    char pad[80];
    pthread_t tid;
    POINT_T *points;
    int num_elems;
    long long SX;
    long long SY;
    long long SXX;
    long long SYY;
    long long SXY;
} lreg_args;
```

ステップ 4: メモリーアクセス解析を再度実行する

図 9 は、新しい解析結果です。

Elapsed Time: 12.764s

CPU Time:	98.007s
Memory Bound:	21.1%

The metric value is high. This can indicate that the significant fraction of execution pipeline slots could be stalled due to demand memory load and stores. Use VTune Amplifier XE Memory Access analysis to have the metric breakdown by memory hierarchy, memory bandwidth information, correlation by memory objects.

L1 Bound:	0.005
L2 Bound:	0.000
L3 Bound:	0.000
DRAM Bound:	0.168

This metric shows how often CPU was stalled on the main memory (DRAM). Caching typically improves the latency and increases performance.

Other:	4.2%
Loads:	413,866,241,595
Stores:	48,737,731,055

9 経過時間

新しい経過時間は 12 秒です。構造体にパディングを追加する 1 行のコード変更で、アプリケーションのパフォーマンスを約 4 倍向上できたことになります。Memory Bound (メモリーバウンド) メトリックも小さくなり、L1 バウンドの問題も解決されました。

困難な問題 2: NUMA 問題

NUMA をサポートするプロセッサーでは、実行している CPU のキャッシュミスを把握するだけでは十分ではありません。NUMA アーキテクチャーでは、別の CPU のキャッシュと DRAM を参照することもできます。このようなアクセスのレイテンシーは、ローカルよりも大きくなります。これらのリモート・メモリー・アクセスを識別して最適化する必要があります。

ここでは、OpenMP* を使用して並列化された単純な Triad (積和演算) アプリケーションを、デュアルソケットのインテル® Xeon® プロセッサー上で実行して検証します。

以下にコードを示します。

```
static double   a[N];
static double   b[N];
static double   c[N];

void doTriad(double x)
{
#pragma omp parallel for
    for (int i = 0; i < N; i++)
        a[i] = b[i] + x*c[i];
}

int main() {

    for (int i = 0; i < N; ++i)
    {
        a[i] = 1.0;
        b[i] = 2.0;
        c[i] = 0.0;
    }

    for (int n = 0; n < NTIMES; ++n)
    {
        doTriad(3.0);
    }

    return 0;
}
```

最初に、配列を初期化し、"omp parallel for" を使用する Triad 関数を呼び出します。

ステップ 1: メモリーアクセス解析を実行する

アプリケーションのメモリーアクセス解析を実行します。このアプリケーションは、DRAM 帯域幅に依存していて、システムの帯域幅を最大限に利用していないことが予想されます (**図 10**)。

Elapsed Time: 12.449s

CPU Time:	635.354s
Memory Bound:	**45.6%**

The metric value is high. This can indicate that the significant fraction of execution pipeline slots could be stalled due to demand memory load and stores. Use VTune Amplifier XE Memory Access analysis to have the metric breakdown by memory hierarchy, memory bandwidth information, correlation by memory objects.

Loads:	234,962,704,886
Stores:	43,997,059,946
LLC Miss Count:	**350,021**
Average Latency (cycles):	17
KNL Bandwidth Estimate (GB/s):	3.129
Total Thread Count:	57
Paused Time:	0s

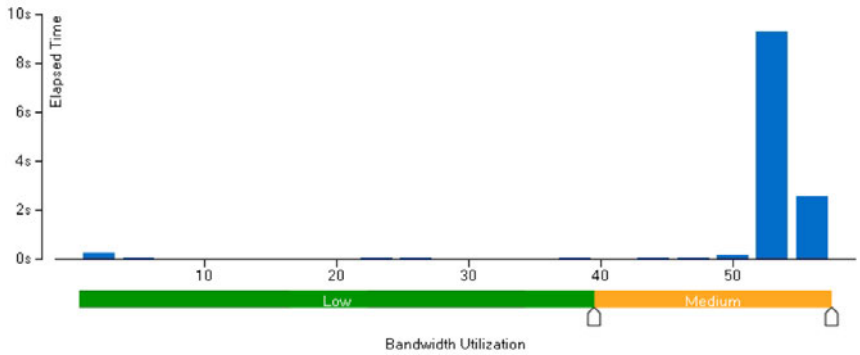
*N/A is applied to metrics with undefined value. There is no data to calculate the metric.

System Bandwidth

Bandwidth Utilization Histogram

This histogram displays a percentage of the wall time the bandwidth was utilized by certain value. Use sliders at the bottom of the histogram to define thresholds for Low, Medium and High utilization levels. You can use these bandwidth utilization types in the Bottom-up view to group data and see all functions executed during a particular utilization type. To learn bandwidth capabilities, refer to your system specifications or run appropriate benchmarks to measure them; for example, Intel Memory Latency Checker can provide maximum achievable DRAM and QPI bandwidth.

Bandwidth Domain: DRAM, GB/sec



10 メモリーアクセス解析

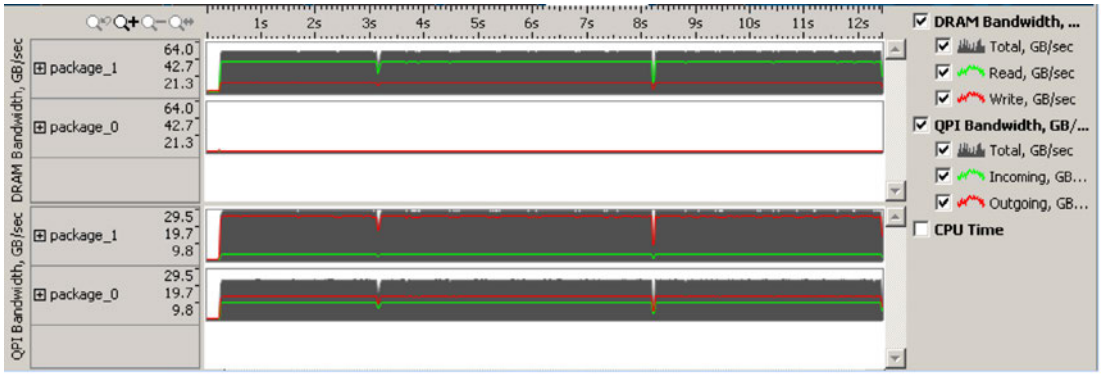
サマリーセクションには、非常に有益なメトリックが含まれています。経過時間は 12.449 秒で、Memory Bound (メモリーバウンド) メトリックは高く、予想どおりハイライトされています。Bandwidth Utilization (帯域幅使用率) ヒストグラムは、DRAM 帯域幅の使用率が 50 ～ 60GB/秒と中程度であることを示しており、調査する必要があります。

その他にも、次のメトリックが役立ちます。

- **Average Latency (平均レイテンシー)。**メモリーアクセスにかかった平均サイクル数。
  注: L1 メモリーアクセスは通常 4 サイクルですが、リモート DRAM アクセスには約 300 サイクルかかります。
- **KNL Bandwidth Estimate (KNL 帯域幅の推定値)。**次世代のインテル® Xeon Phi™ コプロセッサー (開発コード名 Knights Landing [KNL]) 上で実行した場合のコアごとの帯域幅の推定値。このプラットフォームへの移行を検討している場合、コードのメモリーアクセス部分の対応状況を確認するのに役立ちます。

ステップ 2: 帯域幅使用率を調査する

[Bottom-Up (ボトムアップ)] タブに切り替えて (**図 11**) 詳細を確認します。



11 [Bottom-Up (ボトムアップ)] タブ

タイムライン・グラフから、DRAM 帯域幅が 1 つのソケット package_1 でのみ使用されており、QPI (intra-socket) トラフィックが最大 30GB/秒と高いことが分かります。これは、NUMA マシンにおける典型的な問題で、メモリーは 1 つのノードに割り当てられ、ワークは複数のノードで分割されています。この場合、一部のノードはローカル・メモリー・アクセスよりも非常に遅い QPI リンクを利用して、データをリモートでロードしなければなりません。

BLOG HIGHLIGHTS

Knights Landing (開発コード名) 上の MCDRAM (高帯域幅メモリー) の概要

MIKE PEARCE >

次世代のインテル® Xeon Phi™ コプロセッサー x200 製品ファミリー (開発コード名 Knights Landing) には、従来の DDR4 に加えて、新しいメモリー・テクノロジーである MCDRAM (Multi-Channel DRAM) と呼ばれる高帯域幅のオンパッケージ・メモリーが搭載されています。MCDRAM は高帯域幅 (DDR4 の 4 倍以上高速)、低容量 (最大 16GB) のメモリーで、Knights Landing (開発コード名) シリコンに含まれています。MCDRAM は L3 キャッシュ (メモリー・サイド・キャッシュ)、個別の NUMA ノード (割り当て可能なメモリー)、あるいはその中間として構成することができます。異なるメモリーモードでシステムを起動できるため、ソフトウェア側でアプリケーションに最適なモードを理解することは困難です。

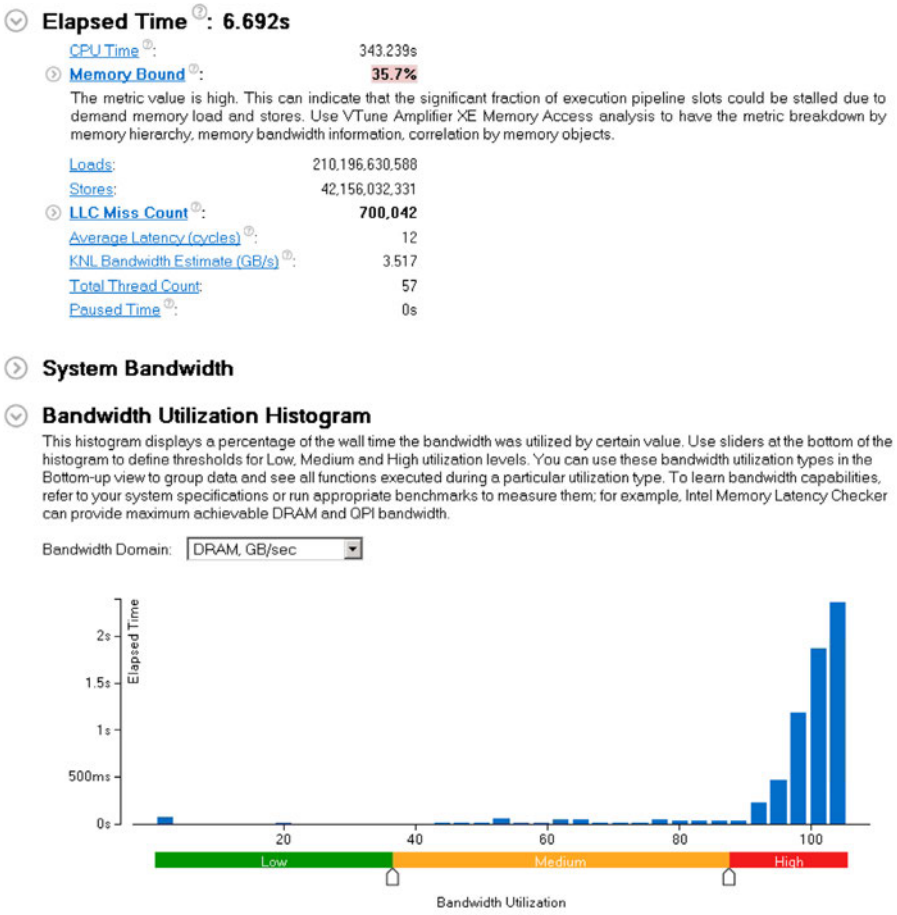
この記事の続きはこちらでご覧になれます。 >

ステップ 3: リモート・メモリー・アクセスを回避するようにコードを変更する

両方のソケットがローカルメモリーにのみアクセスするようにコードを変更して、リモート・ノード・アクセスを回避すれば、より高速に実行できるはずです。Linux* では、メモリーページは最初のアクセス時に割り当てられます。そのため、メモリーアクセスが発生するノードでメモリーを初期化することで、簡単に問題を解決できます。初期化ループに **omp parallel for** プラグマを追加します。

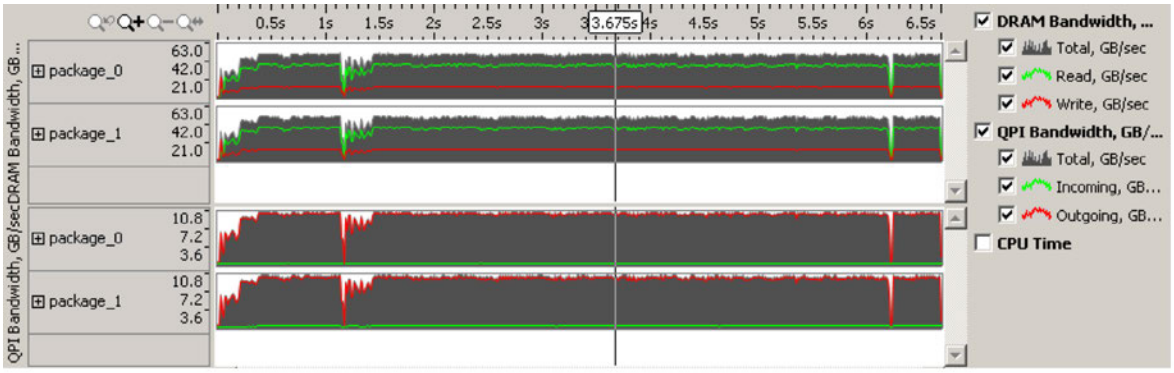
```
static double   a[N];
static double   b[N];
static double   c[N];

void doTriad(double x)
{
#pragma omp parallel for
    for (int i = 0; i < N; i++)
        a[i] = b[i] + x*c[i];
}

int main() {

#pragma omp parallel for
    for (int i = 0; i < N; ++i)
    {
        a[i] = 1.0;
        b[i] = 2.0;
        c[i] = 0.0;
    }

    for (int n = 0; n < NTIMES; ++n)
    {
        doTriad(3.0);
    }

    return 0;
}
```

ステップ 4: KMP_AFFINITY 変数を指定してメモリー解析を再度実行する



12 帯域幅使用率

経過時間が 12.449 秒から 6.69 秒に減りました。これは、ほぼ 2 倍のスピードアップです。また、予想どおり DRAM と帯域幅の使用率が高くなりました (**図 12**)。



13 [Bottom-Up (ボトムアップ)] タブ

帯域幅は 2 つのソケットで等分割され、QPI トラフィックは 1/3 になりました (**図 13**)。

NUMA アーキテクチャーは複雑なため、メモリーアクセスに細心の注意が必要です。アプリケーションの最もレイテンシーが大きいメモリーアクセスを最適化すると、最もパフォーマンスを向上することができます。

困難な問題 3: 次世代のインテル® Xeon Phi™ コプロセッサー向けに最適化する

メモリー帯域幅によりアプリケーションのパフォーマンスが制限されることはよくあります。新しい世代のインテル® Xeon Phi™ コプロセッサーは、物理帯域幅の制限を緩和するため、特殊なオンパッケージの MCDRAM メモリーを搭載しています。MCDRAM メモリーは、高帯域幅を実現します。また、従来の DRAM (DDR4) メモリーも搭載されており、90GB/秒でアクセスできます。ただし、MCDRAM メモリーは容量が小さいため、その利点を最大限に得られるように配置するデータ・オブジェクトを決定することが重要です。

ステップ 1: メモリーアクセス解析を実行する

この例では、Mantevo Suite* の miniFE* ベンチマーク (**mantevo.org/**) を使用します。インテル® VTune™ Amplifier XE のメモリーアクセス解析機能を利用して、ベンチマークを変更せずにそのままプロファイルしたところ、ベンチマーク・コードはメモリー依存であることが分かりました。これは、従来のメモリーでは高トラフィックになり、MCDRAM の高帯域幅により利点が得られる可能性があることを意味します (**図 14**)。

Memory Access Memory Usage viewpoint (change)

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Platform

Elapsed Time: 17.055s

CPU Time: 285.429s

Memory Bound: 47.2%

The metric value is high. This can indicate that the significant fraction of execution pipeline slots could be stalled due to demand memory load and stores. Use VTune Amplifier XE Memory Access analysis to have the metric breakdown by memory hierarchy, memory bandwidth information, correlation by memory objects.

L1 Bound: 0.085

This metric shows how often machine was stalled without missing the L1 data cache. The L1 cache typically has the shortest latency. However, in certain cases like loads blocked on older stores, a load might suffer a high latency even though it is being satisfied by the L1.

L2 Bound: 0.000

L3 Bound: 0.011

DRAM Bound: 0.402

This metric shows how often CPU was stalled on the main memory (DRAM). Caching typically improves the latency and increases performance.

Memory Bandwidth: 0.847

This metric represents a fraction of cycles during which an application could be stalled due to approaching bandwidth limits of the main memory (DRAM). This metric does not aggregate requests from other threads/cores/sockets (see Uncore counters for that). Consider improving data locality in NUMA multi-socket systems.

Memory Latency: 0.130

This metric represents a fraction of cycles during which an application could be stalled due to the latency of the main memory (DRAM). This metric does not aggregate requests from other threads/cores/sockets (see Uncore counters for that). Consider optimizing data layout or using Software Prefetches (through the compiler).

Other: 0.4%

Loads: 204,715,614,145

Stores: 24,055,360,825

LLC Miss Count: 1,416,084,960

Average Latency (cycles): 27

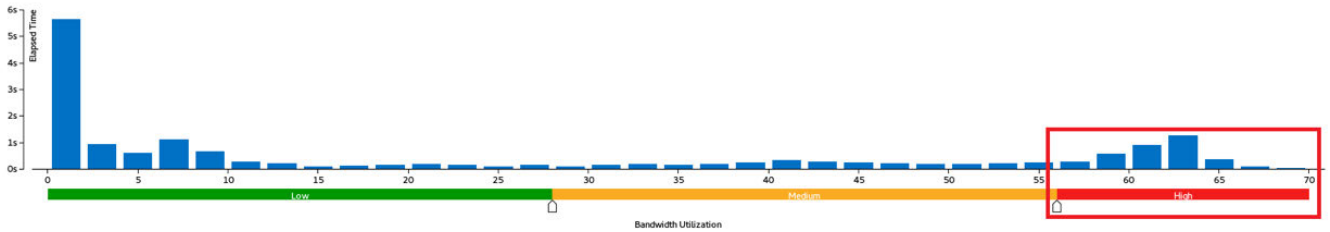
Total Thread Count: 124

Paused Time: 0s

Bandwidth Utilization Histogram

This histogram displays a percentage of the wall time the bandwidth was utilized by certain value. Use sliders at the bottom of the histogram to define thresholds for Low, Medium and High utilization levels. You can use these bandwidth utilization types in the Bottom-up view to group data and see all functions executed during a particular utilization type. To learn bandwidth capabilities, refer to your system specifications or run appropriate benchmarks to measure them; for example, Intel Memory Latency Checker can provide maximum achievable DRAM and QPI bandwidth.

Bandwidth Domain: DRAM, GB/sec



14 メモリーアクセス

ステップ 2: メモリー割り当て関連の帯域幅を調査する

[Bottom Up (ボトムアップ)] ビューで [Function/Memory Object/Allocation Stack (関数/メモリー・オブジェクト/割り当てスタック)] グループをメモリーアクセス解析結果に適用すると、ほとんどのメモリーアクセスは miniFE::CSRMatrix クラスと miniFE::Vector クラスのオブジェクトで発生していることが分かります (**図 15**)。

Grouping: Function / Memory Object Allocation Source / Allocation Stack

Function / Memory Object Allocation Source / Allocation Stack	CPU Time	Memory Bound						Average Latency (cycles)	Loads	Stores
		L1 Bound	L2 Bound	L3 Bound	DRAM Bound: Memory Bandwidth	DRAM Bound: Memory Latency	Other			
miniFE::cg_solve<miniFE::CSRMatrix<double, int, int>, miniFE::Vector	124.646s	0.000	0.002	0.008	1.000	0.000	0.017	83	69,040,207,120	867,013,005
new_allocator.h:104								164	28,990,086,970	1,000,015
__gnu_cxx::new_allocator<double>::allocate ← __gnu_cxx::__all								164	28,990,086,970	1,000,015
new_allocator.h:104								73	24,745,074,235	0
__gnu_cxx::new_allocator<int>::allocate ← __gnu_cxx::__alloc_t								73	24,745,074,235	0
new_allocator.h:104								23	12,040,036,120	840,012,600
__gnu_cxx::new_allocator<double>::allocate ← __gnu_cxx::__all								23	12,040,036,120	840,012,600
new_allocator.h:104								13	3,190,009,570	0
__gnu_cxx::new_allocator<int>::allocate ← __gnu_cxx::__alloc_t								13	3,190,009,570	0

15 [Function/Memory Object/Allocation Stack (関数/メモリー・オブジェクト/割り当てスタック)] グループ

さらに、スタックペインにある miniFE::CSRMatrix クラスデータの完全なコールスタック割り当て情報から、割り当て操作のソースコードの位置を確認できます (**図 16**)。

Loads (Memory Allocation)

Viewing 1 of 8 selected stack(s)

42.0% (28990086970 of 69040207120)

```
miniFE.x!__gnu_cxx::new_allocator<double>::allocate - new_allocator.h
miniFE.x!__gnu_cxx::__alloc_traits<std::allocator<double>>::allocate - alloc_traits.h:182
miniFE.x!std::vector<double, std::allocator<double>>::_M_allocate_and_copy<double*> - stl_vector.h:1224
miniFE.x!miniFE::CSRMatrix<double, int, int>::reserve_space+0x136 - CSRMatrix.hpp:93
miniFE.x!miniFE::generate_matrix_structure<miniFE::CSRMatrix<double, int, int>>+0x13b - generate_matrix_structure.hpp:81
miniFE.x!miniFE::driver<double, int, int>+0x18f - driver.hpp:181
miniFE.x!main+0x881 - main.cpp:178
miniFE.x!_start+0x28 - [unknown source file]
```

16 スタックペイン

CSRMatrix.hpp:93 の行をクリックすると、ソースビューが開き CSRMatrix.hpp ファイルの行 93 にある割り当て操作が表示されます (**図 17**)。

```
89
90    void reserve_space(unsigned nrows, unsigned ncols_per_row)
91    {
92      rows.resize(nrows);
93      row_offsets.resize(nrows+1);
94      packed_cols.reserve(nrows * ncols_per_row);
95      packed_coefs.reserve(nrows * ncols_per_row);
```

17 ソースビュー

miniFE::CSRMatrix クラスデータは、格納されるベクトル要素にカスタム・メモリー・アロケーター・クラスを指定できる、柔軟性のある STL ベクトルコンテナーにより管理されます。

```
template<typename Scalar,
         typename LocalOrdinal,
         typename GlobalOrdinal,
         typename ComputeNode>
struct CSRMatrix {
…
  typedef Scalar        ScalarType;
  typedef LocalOrdinal  LocalOrdinalType;
  typedef GlobalOrdinal GlobalOrdinalType;
  typedef ComputeNode   ComputeNodeType;

  bool                        has_local_indices;
  std::vector<GlobalOrdinal> rows;
  std::vector<LocalOrdinal>  row_offsets;
  std::vector<LocalOrdinal>  row_offsets_external;
  std::vector<GlobalOrdinal> packed_cols;
  std::vector<Scalar>        packed_coefs;
  LocalOrdinal               num_cols;
  ComputeNode&               compute_node;
…
}
```

ステップ 3: 高帯域幅メモリーを使用してオブジェクトを割り当てる

memkind ライブラリー API (**https://github.com/memkind/memkind**) を利用して、miniFE::CSRMatrix と miniFE::Vector データ・オブジェクトを MCDRAM メモリーに移動します。hbwmalloc.h ヘッダーファイルは、MCDRAM (高帯域幅) メモリーで STL ベクトルコンテナーのパラメーター化に使用される hbwmalloc::hbwmalloc_allocator クラスの実装を提供します。

ここでは、次のようにコードを変更します。

```
…
#include “/opt/mk-0.3.0/include/hbwmalloc.h”
…
template<typename Scalar,
         typename LocalOrdinal,
         typename GlobalOrdinal,
         typename ComputeNode>
struct CSRMatrix {
…
  typedef Scalar        ScalarType;
  typedef LocalOrdinal  LocalOrdinalType;
  typedef GlobalOrdinal GlobalOrdinalType;
  typedef ComputeNode   ComputeNodeType;

  bool                        has_local_indices;
  std::vector<GlobalOrdinal, hbwmalloc::hbwmalloc_allocator<GlobalOrdinal> > rows;
  std::vector<LocalOrdinal, hbwmalloc::hbwmalloc_allocator<GlobalOrdinal> >  row_offsets;
  std::vector<LocalOrdinal, hbwmalloc::hbwmalloc_allocator<GlobalOrdinal> >  row_offsets_
external;
  std::vector<GlobalOrdinal, hbwmalloc::hbwmalloc_allocator<GlobalOrdinal> > packed_cols;
  std::vector<Scalar, hbwmalloc::hbwmalloc_allocator<GlobalOrdinal> >        packed_coefs;
  LocalOrdinal               num_cols;
  ComputeNode&               compute_node;
…
}
```

変更したソースコードをリビルドして、メモリーアクセス解析を再度実行すると、miniFE::CSRMatrix と miniFE::Vector データ・オブジェクトが memkind ライブラリーの hbwmalloc::hbwmalloc_allocator クラスによって作成されていることが分かります (**図 18**)。

Grouping: Function / Memory Object Allocation Source / Allocation Stack

Function / Memory Object Allocation Source / Allocation Stack	CPU Time	Memory Bound						Average Latency (cycles)	Loads	Stores
		L1 Bound	L2 Bound	L3 Bound	DRAM Bound		Other			
					Memory Bandwidth	Memory Latency				
miniFE::cg_solve<miniFE::CSRMatrix<double, int, int>, miniFE::Ve	104.645s	0.011	0.000	0.013	1.000	0.000	0.015	75	68,645,205,935	873,013,095
hbwmalloc.h:143								171	32,600,097,800	4,000,060
hbwmalloc::hbwmalloc_allocator<double>::allocate ← __gnu								171	32,600,097,800	4,000,060
hbwmalloc.h:143								49	21,790,065,370	0
hbwmalloc::hbwmalloc_allocator<int>::allocate ← __gnu_cxx								49	21,790,065,370	0
hbwmalloc.h:143								23	11,020,033,060	842,012,630
hbwmalloc::hbwmalloc_allocator<double>::allocate ← __gnu								23	11,020,033,060	842,012,630
hbwmalloc.h:143								7	3,125,009,375	0
hbwmalloc::hbwmalloc_allocator<int>::allocate ← __gnu_cxx								7	3,125,009,375	0

18 メモリーアクセス解析

ステップ 4: ベンチマークを再度実行する

変更したバージョンのベンチマークをインテル® Xeon Phi™ コプロセッサー上の MCDRAM メモリーを利用して実行すると、従来の DRAM メモリーに利用頻度の高いデータを割り当てるオリジナルバージョンのほぼ 4 倍高速になりました。

```
[vtune@nntvtune46 src]$ /usr/bin/time /tmp/miniFE-2.0_openmp_ref_ORIG/src/miniFE.x.sh
MiniFE Mini-App, OpenMP Peer Implementation
Creating OpenMP Thread Pool...
Counted: 12 threads.
Running MiniFE Mini-App...
      creating/filling mesh...0.197327s, total time: 0.197329
generating matrix structure...13.5858s, total time: 13.7832
         assembling FE data...13.3513s, total time: 27.1345
      imposing Dirichlet BC...2.61192s, total time: 29.7464
      imposing Dirichlet BC...1.11535s, total time: 30.8617
making matrix indices local...1.19209e-06s, total time: 30.8617
Starting CG solver ...
Initial Residual = 201.001
Iteration = 20   Residual = 0.0609161
...
Iteration = 200   Residual = 0.00112011
Final Resid Norm: 0.00112011
2671.76user 9.53system 3:55.02elapsed 1140%CPU (0avgtext+0avgdata 1511308maxresident)
k0inputs+8outputs (0major+49614minor)pagefaults 0swaps

[vtune@nntvtune46 src]$ /usr/bin/time /tmp/miniFE-2.0_openmp_ref_KNL/src/miniFE.x.sh
MiniFE Mini-App, OpenMP Peer Implementation
Creating OpenMP Thread Pool...
Counted: 12 threads.
Running MiniFE Mini-App...
      creating/filling mesh...0.198685s, total time: 0.198686
```

```
generating matrix structure...13.9371s, total time: 14.1358
         assembling FE data...13.1823s, total time: 27.3181
      imposing Dirichlet BC...2.51502s, total time: 29.8331
      imposing Dirichlet BC...1.10896s, total time: 30.942
making matrix indices local...9.53674e-07s, total time: 30.942
Starting CG solver ...
Initial Residual = 201.001
Iteration = 20   Residual = 0.0609161
…
Iteration = 200   Residual = 0.00112011
Final Resid Norm: 0.00112011
475.87user 2.09system 0:52.25elapsed 914%CPU (0avgtext+0avgdata 2598752maxresident)
k0inputs+8outputs (0major+23143minor)pagefaults 0swaps
```

この例は、帯域幅依存のコードは、最新のインテル® Xeon Phi™ コプロセッサー・プラットフォームで利用可能な MCDRAM メモリーに利用頻度の高いデータを配置することで利点が得られ、memkind ライブラリーを利用することでその作業を簡単に行うことができることを示しています。

まとめ

プログラムのメモリーアクセスを最適化することは極めて重要です。インテル® VTune™ Amplifier XE のようなツールを使って、プログラムがどのようにメモリーにアクセスしているか理解することは、ハードウェア機能を最大限に利用するのに役立ちます。

ここでは、インテル® VTune™ Amplifier XE のメモリーアクセス解析機能の概要を紹介し、この機能を使っていくつかの困難なメモリーの問題を解決する方法を示しました。

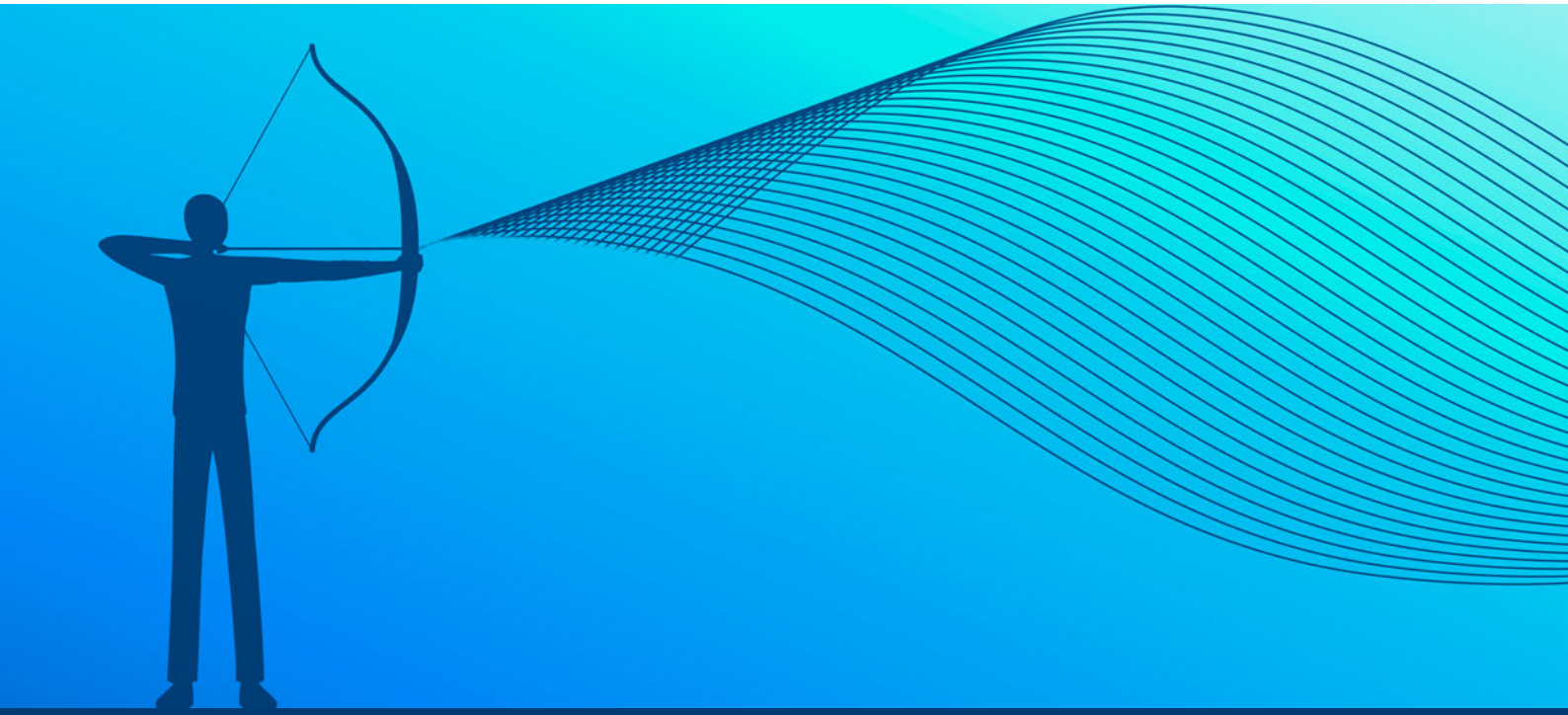
比較的小さなメモリー・オブジェクトで平均レイテンシーの値が高い場合にフォルス・シェアリング問題を検出する方法を紹介し、パディングを追加するたった 1 行のコード変更によりアプリケーションのパフォーマンスを 4 倍向上しました。

大量のリモート・メモリー・アクセスを含む NUMA 問題を検出する方法を説明し、それらのリモートアクセスを排除することでアプリケーションのパフォーマンスを 2 倍向上しました。

最後に、最新のインテル® Xeon Phi™ コプロセッサー・プラットフォームで利用可能なメモリー・テクノロジーにより利点が得られるコード領域を特定することで、ベンチマーク・アプリケーションを 4 倍スピードアップできることを示しました。

インテル® VTune™ Amplifier XE を評価する

インテル® Parallel Studio XE に含まれます >



インテル® IPP を利用した画像識別の最適化

Tencent* の MD5 画像識別スピードが 2 倍に向上

Yueqiang Lu インテル APAC R&D Ltd. アプリケーション・エンジニア
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

Tencent, Inc. は、中国におけるインターネット・サービス・ポータルの最大手です。中国最大のオンライン・ゲーム・コミュニティーと Web ポータル (**qq.com**) に加え、トップ 2 つのアプリケーション (WeChat*、QQ*) を所有しています。

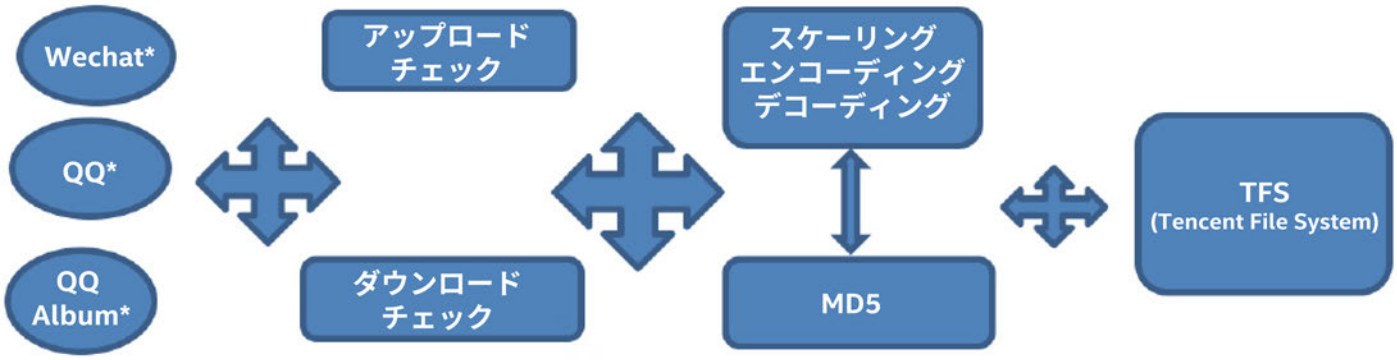
毎日、Tencent* では WeChat*、QQ*、QQ Album* でユーザーにより生成された数十億もの画像を処理しています。人気の高いアプリケーションでは、1 日で数億もの画像がアップロード、格納、処理、そしてダウンロードされます—そのため、大量の計算リソースを必要とします。

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

これらの画像を管理、格納、処理するため、Tencent* は Tencent File System (TFS) を開発しました。しかし、画像サイズは圧縮しても数百ペタバイトに達します。また、爆発的な成長が続いており、クラスターは 20,000 を超えるサーバーにより構成されています。

技術背景

TFS ベースの画像処理システムは、アップロード、エンコード、ダウンロード・サービスを提供します。画像アップロードでは、異なる解像度にスケーリングし、メッセージ・ダイジェスト・アルゴリズム 5 (MD5) を用いて関連 ID を生成します。[1] そして、格納用に画像を WebP* 形式に変換します。画像のダウンロード中、システムは画像の正しい読み取り位置を特定し、ユーザーが要求する形式と解像度に変換する必要があります (**図 1**)。



1 Tencent File System* の画像処理

毎秒、膨大な数の訪問者が Web サイトを訪れるため、ダウンロード・コンポーネントが誤った画像を読み取る可能性がわずかですがあります。このようなエラーを回避するには、MD5 による計算とチェックが必要です。ただし、これは非常に大きな計算ワークロードであるため、Tencent* は MD5 の計算パフォーマンスを最大限に高める必要がありました。

元々、Tencent* は Operator* OS と md5sum* ユーティリティー・ツールを使用して、各画像ファイルの MD5 値を計算していました。インテルは Tencent* のエンジニアと協力して、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**) を利用してパフォーマンスの最適化に取り組みました。その結果、Tencent* はインテル® アーキテクチャー・ベースのプラットフォームで 2 倍のパフォーマンス向上を達成しました。

インテル® ストリーミング SIMD 拡張とソフトウェアの最適化

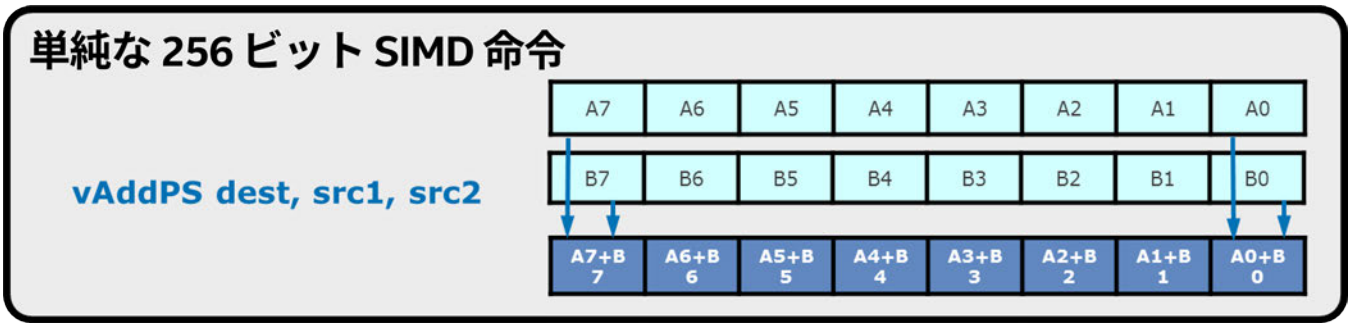
インテル® Pentium® III プロセッサーで、インテル® ストリーミング SIMD 拡張 (インテル® SSE) 命令セットが導入されました。これは、インテル® Pentium® プロセッサーで導入された MMX® 命令と呼ばれる以前の **SIMD** (Single Instruction Multiple Data) 命令セットを大幅に変更したものです。

その後、インテル® アーキテクチャーの進化とともにインテル® SSE 命令セットも進化し、ベクトル幅が広くなり、新しい拡張可能な構文と豊富な機能が追加されました。最新の SIMD 命令セットであるインテル® アドバンスト・ベクトル・エクステンション 2 (インテル® AVX2) は、第 4 世代インテル® Core™ プロセッサー以降で利用できます。

TFS システムで使用されているほとんどのインテル® Xeon® プロセッサーは、インテル® SSE2 をサポートしています。インテル® SSE2 は、インテル® SSE3、インテル® SSE4.x、およびインテル® AVX により補完されます。

インテル® AVX は、インテル® SSE の 256 ビット命令セット拡張で、計算負荷の高いアプリケーションでより優れたパフォーマンスを提供できるように設計されています。インテル® SSE ベースの SIMD 命令セットに浮動小数点演算と整数演算の新しい機能が追加されており、よりコンパクトな SIMD 命令セットが含まれます。

図 2 は、1 つの SIMD 操作で 8 つのデータ (32 ビットの整数型、浮動小数点型) 命令を処理できることを示しています。



2 8 つのデータ命令を処理する SIMD 操作

インテル® AVX は、浮動小数点と整数データのベクトル処理の幅を広げることで、パフォーマンスを向上します。その結果、画像とオーディオ/ビデオ処理、科学シミュレーション、金融解析、3D モデリング/解析のような広範なアプリケーションで、ハイパフォーマンスと効率良いデータ管理を実現します。

インテル® SSE により利点が得られるアルゴリズム

インテル® SSE により利点が得られるアルゴリズムには、[2] 単一の 32 ビット/64 ビット・ワードよりも大きなデータセットに対して論理演算や数値演算を行うアルゴリズムが含まれます。インテル® SSE はベクトル命令または SIMD アーキテクチャーを利用して、ビット単位の XOR (排他的論理和)、整数/浮動小数点の累乗算、単一クロックサイクルでの複数の 32 ビット/64 ビット・ワードのスケーリングなどの操作を完了します。スピードアップは、各算術演算子や論理演算子に適用される並列演算とベクトルサイズ (マルチワード・データ) によってもたらされます。

以下は、SIMD ベクトル命令により大きな利点が得られるアルゴリズムの例です。

- **画像処理とグラフィックス。**どちらも解像度 (単位面積当たりの画素数) とピクセル・エンコーディング (明るさと色を表す 1 画素当たりのビット数) がスケーリングし、処理フレームレートが向上します。
- **デジタル信号処理 (DSP)。**センサーや計器からデジタル化されたサンプルは、画像のような解像度とデータ取得レートになります。通常、1 次元のデジタル化されたデータの時系列は、大量の時系列サンプルを処理する DFT (離散フーリエ変換) などのアルゴリズムを使用して変換されます。
- **ダイジェスト、ハッシュ、エンコード。**単純なパリティー、巡回冗長検査 (CRC)、MD5、セキュア・ハッシュ・アルゴリズム (SHA)、ガロア演算、リードソロモン・エンコーディング、暗号ブロックチェーン (CBC) のようなセキュリティー、データ破損防止、データ損失防止に使用されるアルゴリズムはすべて、データブロック (通常、数キロバイト) に対して論理演算子または算術演算子を利用します。
- **データ変換とデータ圧縮。**ほとんどの場合、エンジニアリング・シミュレーションや科学計算には経時的なデータ変換がつきもので、変換されるデータグリッドが含まれます。例えば、物理熱力学、力学、流体力学、電界モデルでは、浮動小数点値のグリッドで有限要素の物理フィールドを表します。これらの有限要素グリッドは、物理プロセスをシミュレーションするため、経時的な数学的変換によって更新されます。

インテル® AVX 向けに最適化されたインテル® IPP

インテル® IPP は、あらゆる画像/信号処理、データ圧縮、暗号化のニーズに適したパフォーマンス・ビルディング・ブロックです。これらの関数はロイヤルティー・フリーで、インテル® SSE、インテル® AVX、インテル® AVX2、およびインテル® AVX-512 命令セットを使用して高度に最適化されているため、最適化コンパイラー単独で生成した場合よりも高速なアプリケーションを生成できます。[3]

表 1 にインテル® IPP の機能を要約します。

パフォーマンスと電力効率を最適化	インテルにより開発され、 将来にわたって開発期間を短縮	広範なクロスプラットフォームと OS 機能
高度にチューニングされたルーチン	現在および以前のプロセッサー向けに最適化	高度に最適化された数千の信号、データ、メディア関数
インテル® SSE4、インテル® SSSE3、インテル® SSE、インテル® AVX、インテル® AVX2、インテル® AVX-512 命令セットを使用して高度に最適化	開発、デバッグ、保守期間を短縮	広範なドメインをサポート
最適化コンパイラー単独で生成したよりも高速	一度記述したコードで将来の最適化にも対応	インテル® Quark™ 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™ コプロセッサー・プラットフォームをサポート

表 1. インテル® IPP の機能

AVX = アドバンスト・ベクトル・エクステンション命令
SSE = ストリーミング SIMD 拡張命令
SSSE = ストリーミング SIMD 拡張命令補足命令

インテル® IPP ライブラリーは、さまざまな SIMD 命令セット向けに最適化されています。さらに、インテル® IPP は、実行しているプロセッサーで利用可能な SIMD 命令を判定して、そのプロセッサーに最適な SIMD 命令を選択する自動「ディスパッチ」機能も提供します。

表 2 は、インテル® IPP で使用されるプロセッサー固有コードです。

プロセッサー固有ライブラリー		
IA-32 アーキテクチャー	**インテル® 64 アーキテクチャー**	**説明**
px	mx	インテル® ストリーミング SIMD 拡張 (インテル® SSE) 対応プロセッサー向けに最適化された汎用コード
w7		インテル® SSE2 対応プロセッサー向けに最適化されたコード
	m7	インテル® SSE3 対応プロセッサー向けに最適化されたコード
v8	u8	インテル® Atom™ プロセッサーを含む、インテル® ストリーミング SIMD 拡張命令 3 補足命令 (インテル® SSSE3) 対応プロセッサー向けに最適化されたコード
p8	y8	インテル® SSE4.1 対応プロセッサー向けに最適化されたコード
g9	e9	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) とインテル® AES New Instructions (インテル® AES-NI) 対応プロセッサー向けに最適化されたコード
h9	l9	インテル® AVX2 対応プロセッサー向けに最適化されたコード
	n0	Knights Landing (開発コード名) 上のインテル® AVX2-512 (F、CD、ER、PF) 向けに最適化されたコード
	k0	インテル® Xeon® プロセッサー上のインテル® AVX2-512 (F、CD、BW、DQ、VL) 向けに最適化されたコード

表 2. プロセッサー固有コード

ディスパッチについては、「**インテル® IPP ライブラリーの CPU ディスパッチの理解**」(英語) を参照してください。インテル® AVX 向けに最適化されたインテル® IPP 関数については、「**インテル® AVX 向けに最適化されたインテル® IPP 関数**」(英語) を参照してください。

インテル® IPP の MD5

ハッシュ関数は、デジタル署名とともに暗号化に使用されます。また、データの整合性を確保するためにも使用されます。デジタル署名とともに使用する場合は、一般公開されている関数でメッセージをハッシュし、結果のハッシュ値に署名します。メッセージを受け取った側は、メッセージをハッシュし、ブロックサイズとハッシュ値を照合します。

ハッシュ関数は、「メッセージ・ダイジェスト」や「一方向暗号化関数」とも呼ばれます。データの整合性を確保する場合は、ハッシュ関数を使用して特定の入力に対応するハッシュ値を計算します。そして、必要に応じて、入力データが変更されないことをチェックします。また、利用可能な入力を使用して再度ハッシュ値を計算し、オリジナルのハッシュ値と比較します。インテル® IPP は、ストリーミング・メッセージで次のハッシュ・アルゴリズムを実装しています。

- MD5 [RFC 1321]
- SHA-1
- SHA-224
- SHA-256
- SHA-384
- SHA-512 [FIPS PUB 180-2]

これらのアルゴリズムは、エンタープライズ・アプリケーションで広範に利用されています。

MD5 について

MD5 は、一般に利用されている暗号化ハッシュ関数で、通常 32 桁の 16 進数をテキスト形式で表現した 128 ビット (16 バイト) のハッシュ値を生成します。

MD5 は、要件の厳しい環境では「暗号化に適していない」と見なされていましたが、ソフトウェアの世界では、転送したファイルが受信者に到達したことを確認する手段として広く利用されてきました。例えば、ファイルサーバーは、ユーザーがダウンロードしたファイルのチェックサムと比較できるように、ファイルの事前に計算された MD5 (md5sum) チェックサムを提供することがよくあります。ほとんどの Linux* ディストリビューションには、md5sum ユーティリティーが含まれています。

インテル® IPP の MD5 関数は、ハッシュ・アルゴリズムをストリーミング・メッセージのダイジェストに適用します。チェーン・ダイジェスト値の計算を管理するため、ステート・コンテキスト (ippsSHA1State など) を使用してすべての必要な変数を保持します。例えば、MD5 ハッシュ・アルゴリズムを実装するプリミティブは、ippsMD5State コンテキストを使用する必要があります。関数 Init (MD5Init) は、コンテキストを初期化し、指定された初期化ベクトルを設定します。初期化後、関数 Update (MD5Update) は、選択されたハッシュ・アルゴリズムを利用して、すべてのメッセージブロックがなくなるまで、入力メッセージストリームをダイジェストします。関数 Final (MD5Final) は、指定されたパディングスキームにより、不完全なメッセージブロックにパディングを追加して、最終メッセージブロックにします。そして、ハッシュ・アルゴリズムを利用して、最終ブロックをメッセージ・ダイジェスト値に変換します。

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

次の例は、アプリケーション・コードで MD5 関数を利用して、入力メッセージストリームをダイジェストする方法を示します。

1. 関数 MD5GetSize を呼び出し、ipps MD5State コンテキストの設定に必要なサイズを取得します。
2. 必要なメモリー空間を割り当てます。メモリーを割り当てたら、MD5Init 関数を呼び出し、MD5 で指定された初期化ベクトルを利用して、コンテキストの初期ステートを設定します。
3. キューにある入力メッセージストリームがなくなるまで、関数 MD5Update を呼び出してダイジェストします。現在のダイジェスト値を特定するため、2 つの MD5Update 呼び出しの間に MD5GetTag を呼び出します。
4. 関数 MD5Final を呼び出し、不完全なブロックにパディングを追加して最終 MD5-1 メッセージブロックにし、160 ビットのメッセージ・ダイジェスト値に変換します。
5. コンテキストに格納されている秘密データをクリーンアップします。
6. オペレーティング・システムのメモリー解放サービス関数を呼び出し、ippsMD5State コンテキストを解放します。

インテルのエンジニアは、主にベクトル化、インテル® SSE 命令、インテル® アーキテクチャーの機能 (キャッシュ効率やレジスター再利用など) によりインテル® IPP 関数を最適化しました。

インテル® IPP の MD5 実装では、次の最適化手法が利用されています。

- 小さなループの代わりに、完全にアンロールされたコードを使用
- メモリー操作の代わりに、レジスターによる巡回置換を使用
- 一般的なパラメーター化された 32 ビットの回転の代わりに、コードで直接回転

「インテルのエンジニアとの共同作業で、弊社のオンライン画像格納・処理アプリケーションの画像識別コンポーネントにインテル® IPP ライブラリーを採用しました。その結果、アプリケーションのパフォーマンスが大幅に向上し、運用コストも大幅に削減されました。インテルの協力に感謝しています。今後も、機会があれば共同作業を行っていきたいと思います。」

TFS ベースの画像格納・処理チームリーダー
Nicholas 氏

インテル® IPP コードで md5sum を置き換えました。**表 3** のコードを使用するだけで、手動による最適化は不要でした。

md5sum コード	インテル® IPP の MD5

`md5sum  performance.PNG > hash.md5`
`cat hash.md5`
`0e5e74555b68db366c85b1b194f258fe performance.PNG.`
 	
`The md5sum is along with Cent OS distribution.`
	`char* intel_md5sum(const char *p, unsigned uiLen)`
	`{`
	`static Ipp8u MD[32];`
	`int size;`
	`IppsMD5State *ctx;`
	`ippsMD5GetSize(&size);`
	`ctx = (IppsMD5State*)malloc(size);`
	`IppStatus st = ippsMD5Init(ctx);`
	`st = ippsMD5Update((const Ipp8u *) p, (int)uiLen, ctx);`
	`st = ippsMD5Final(MD, ctx);`
	`free(ctx);`
	`return (char*)MD;`
	`}`

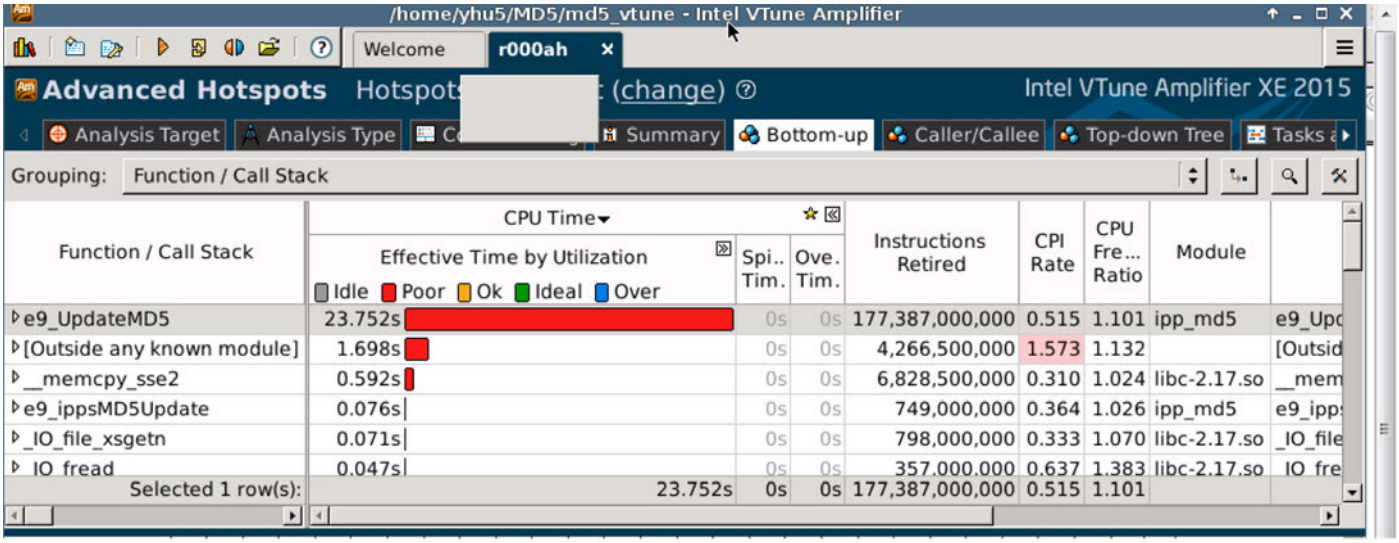
表 3. コード例

インテル® IPP による MD5 の高速化

インテル® IPP の MD5 コード **`md5test.cpp`** は、次のように gcc でコンパイルします。

```
[root@localhost code]# make -f Makefile.gcc
g++ -O2 ipp_md5.cpp -o ipp_md5
-I/opt/intel/compilers_and_libraries_2016.0.109/linux/ipp/include
/opt/intel/compilers_and_libraries_2016.0.109/linux/ipp/lib/intel64/libippcp.a
/opt/intel/compilers_and_libraries_2016.0.109/linux/ipp/lib/intel64/libippcore.a
```

これにより、インテル® IPP 暗号化ライブラリーがプログラムにリンクされ、自動的に計算リソースのパフォーマンスが引き出されます。**図 3** は、インテル® VTune™ Amplifier XE で ipp_md5 プログラムを実行した結果です。e9 (インテル® AVX 向けに最適化された) コードを実行するインテル® IPP 関数 e9_ippsMD5Update で、プログラムの CPU 時間のほとんどが費やされていることが分かります。



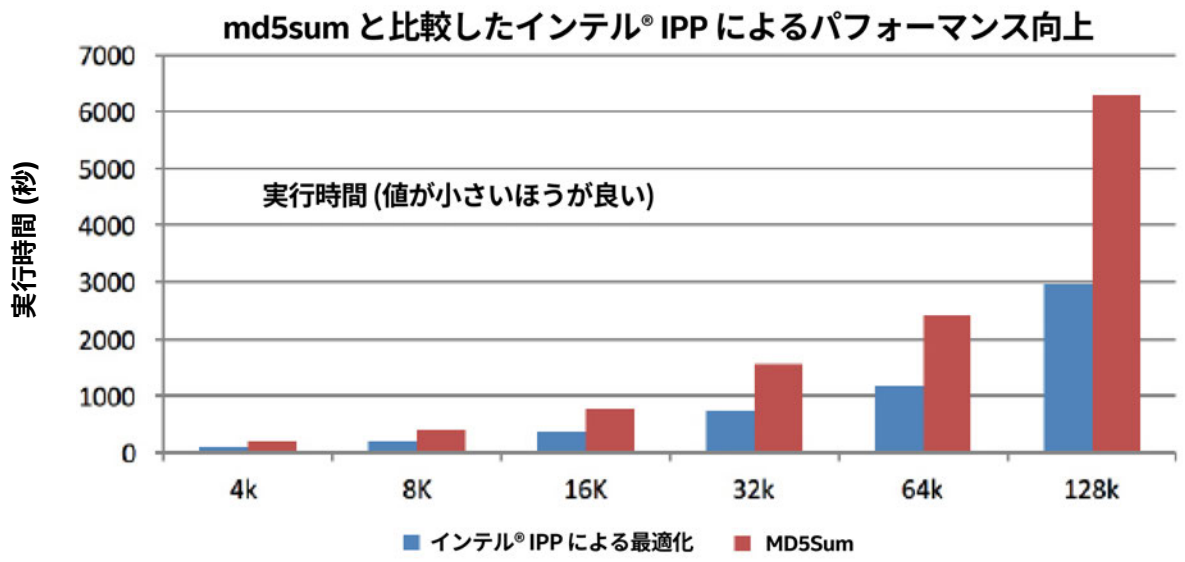
3 インテル® VTune™ Amplifier XE で ipp_md5 を実行した結果

パフォーマンス・データ

テストは、インテル® IPP と Linux* の md5sum を使用して、異なるサイズの画像ファイルで実行しました。10,000 反復のパフォーマンス結果が**表 4** と**図 4** です。

プロセッサー/OS	テストした画像サイズ	md5dum (10,000 反復の平均時間)	IPP_Md5 (10,000 反復の平均時間)
インテル® Xeon® プロセッサー E5-2620 (15M キャッシュ、2GHz、7.20GT/秒、インテル® QPI、インテル® AVX をサポートする CentOS* 6.5)	4K	206ms	95ms
	8K	406ms	189ms
	16K	789ms	369ms
	32K	1574ms	740ms
	64K	2420ms	1183ms
	128K	6273ms	2943ms

表 4. テスト実行のパフォーマンス結果



システム構成 - バージョン: インテル® IPP 9.0.0、ハードウェア: インテル® Xeon® プロセッサー E5-2620 (15M キャッシュ、2GHz、7.20GT/秒、インテル® QPI)、メモリー: 64G、オペレーティング・システム: CentOS* 8 6.5 x86_64、ベンチマークのソース: テスト用画像ファイル。
* 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。

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。

4 テスト実行の結果

インテル® Xeon® プロセッサー E5-2620 (15M キャッシュ、2GHz、7.20GT/秒、インテル® QuickPath インターコネクト [インテル® QPI]、インテル® AVX 対応) では、Linux* の md5sum と比較してパフォーマンスが 100% 向上しました。Tencent* エンジニアは、オンラインシステムにもインテル® IPP の MD5 を実装しました。その結果、オリジナルの MD5 と比較して、パフォーマンスが 60% 向上しました。

まとめ

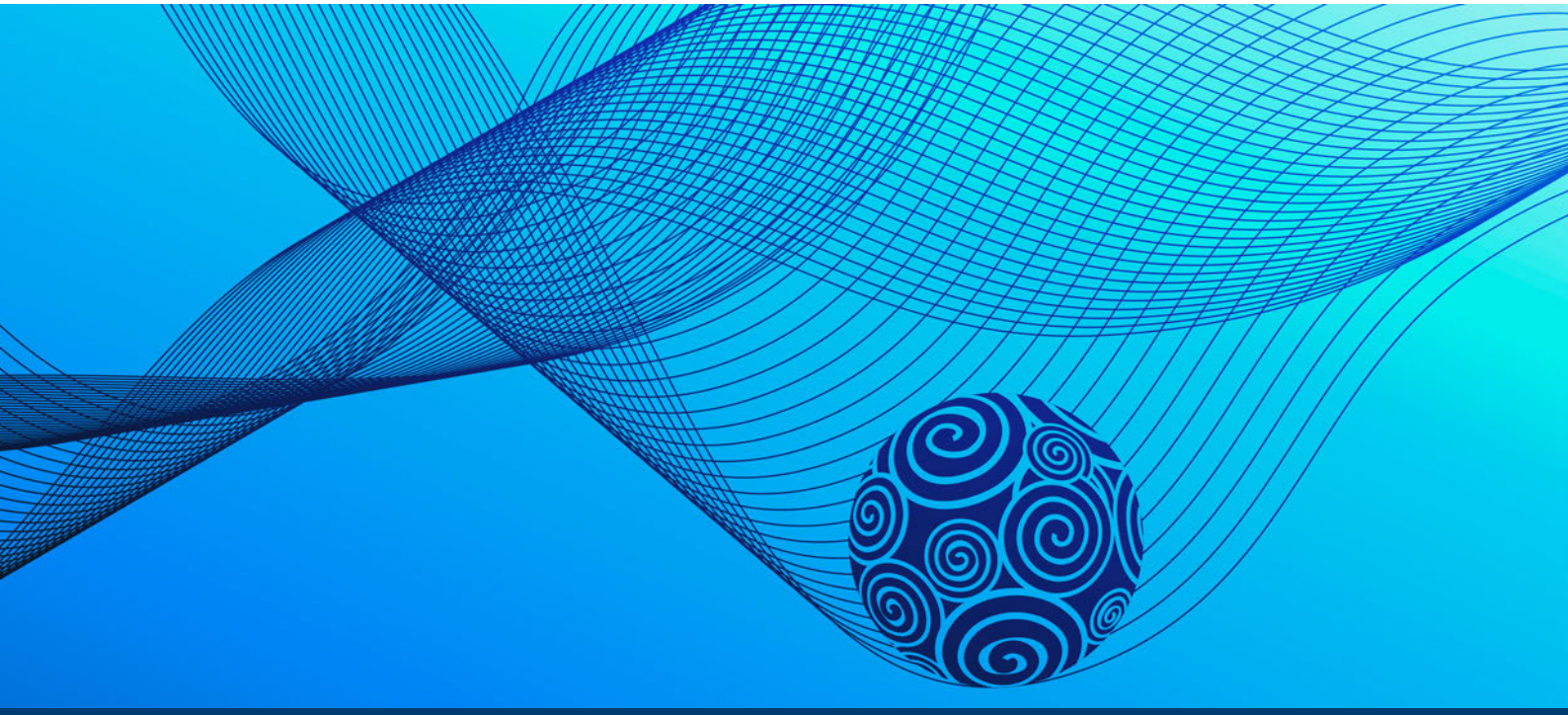
Tencent* では、毎日、WeChat*、QQ*、QQ Album* でユーザーにより生成された数十億もの画像を処理しています。すべての画像は、TFS ベースの画像格納・処理システムによって処理されます。各イメージには、MD5 ハッシュにより一意の ID を付ける必要があります。インテルは Tencent* のエンジニアと協力して、インテル® IPP を利用することで、この関数コンポーネントを最適化し、2 倍のパフォーマンス向上を達成しました。

インテル® IPP を利用することで、画像の md5sum 計算速度を簡単に向上できました。この作業を通じて、インテル® IPP とパフォーマンス・チューニング手法を用いて最新のインテル® ハードウェア向けに最適化することで、これらの計算負荷の高い処理に対応できたことは大きな前進です。

参考文献

1. **Wikipedia* の md5sum の説明**
2. **インテル® ストリーミング SIMD 拡張命令とインテル® IPP によるアルゴリズムの高速化 (英語)**
3. **インテル® AVX 命令を使用した complex float データ型の IIR フィルターの実装**

インテル® IPP をダウンロードする >



最新の IoT と組込みテクノロジーを利用したスマートな開発

インテル® System Studio で優れたアイデアを優れた製品に

Noah Clemons インテル コーポレーション 並列プログラミング製品 テクニカル・コンサルティング・エンジニア

優れたアイデアを優れた製品にするには多くの作業を行う必要がありますが、作業に適したツールを利用すると、そのプロセスは非常に簡単になります。インテルは、組込みシステム開発のニーズをすべて満たすツールスイートとして、インテル® マイクロコントローラー、IoT (Internet of Things) デバイス、組込みプラットフォームのコーディング、解析、デバッグ用の**ソフトウェア開発ツール**を統合した、インテル® System Studio およびインテル® System Studio for Microcontrollers を提供しています。

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

この記事では、次の項目について説明します。

- インテル® System Studio とインテル® System Studio for Microcontroller のコンポーネントの概要
- すべてのプラットフォームでコンポーネントがどのように動作するか
- 新しいマイクロコントローラー・プラットフォームを提供するためにコンポーネントを変更した場所
- インテル® System Studio がどのように IoT のニーズと組込み開発に対応しているか

インテル® System Studio の対象ユーザー

インテル® System Studio の対象ユーザーは次のとおりです。

- プラットフォーム用の正しいツールを必要としているデバイスメーカー
- 既存のプラットフォームに依存し、完全なシステム・ソフトウェア・スタックを確立する必要があるシステム・インテグレーター
- 専用アプリケーションをビルドおよび最適化する必要がある組込みソフトウェア開発者

優れた製品のビルド

インテル® System Studio は、インテル® ハードウェアで動作する製品を開発するために必要なすべてのツールを提供することを目的としており、インテル® Quark™ SoC X1000 やインテル® Atom™ プロセッサー・ベースの IoT ゲートウェイから、インテル® Core™ プロセッサーおよびインテル® Xeon® プロセッサー・ベースのサーバーまで、さまざまな組込みプラットフォームをサポートしています。

インテル® System Studio for Microcontrollers は、インテル® Quark™ マイクロコントローラー D1000/D2000 およびインテル® Quark™ SE マイクロコントローラー向けにカスタマイズされた開発環境を提供するため特別に用意されたツールスイートです。

BLOG HIGHLIGHTS

デバイスの選択

ALEX KATRANOV >

この記事は、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) 4.4 Update 2 以降で利用可能な新しいノード opencl_node について説明するシリーズのパート 3 です。このノードは、インテル® TBB のフローグラフで OpenCL* デバイスの利用と連携を可能にします。このシリーズのパート 1 はこちらからご覧いただけます。

パート 2 では、基本インターフェイスを説明しました。パート 3 では、カーネルの実行に使用するデバイスの選択について述べます。

この記事の続きはこちらでご覧になれます。 >

表 1 に、これらのスイートで提供される機能の一部を示します。

	インテル® System Studio	インテル® System Studio for Microcontrollers
プラットフォーム	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、インテル® Quark™ SoC U シリーズ	インテル® Quark™ マイクロコントローラー D シリーズ
ソフトウェア開発環境	Eclipse* ベースの統合開発環境 (IDE)、コマンドライン、インテル® Graphics Performance Analyzers	Eclipse* ベースの IDE、コマンドライン
ホストシステム	Linux*、Windows*、OS X*[1]	Linux*、Windows*
コンパイラー	インテル® C++ コンパイラー	インテル® C++ コンパイラーまたは GNU* C コンパイラー[2]
ターゲット・プラットフォーム OS	Linux*、Android*、Windows*、FreeBSD*、VxWorks*	ベアメタル、リアルタイム OS
ターゲット・プラットフォーム・ソフトウェア	サンプル、デバッガー、プロファイラー対応ドライバー	ボード対応パッケージ、インテル® Quark™ マイクロコントローラー・ソフトウェア・インターフェイス、サンプル
ライブラリー	インテル® MKL、インテル® IPP、インテル® TBB (画像、信号処理、データ処理、マルチスレッド)	C ランタイム、浮動小数点エミュレーション、DSP ライブラリー
アナライザー	インテル® VTune™ Amplifier for Systems、インテル® Energy Profiler、インテル® Inspector for Systems (メモリー・アナライザー)	電力アナライザー[3]
デバッガー	アプリケーションおよび OS、WinDbg カーネルデバッガー、インテルの GDB 拡張、インテル® System Debugger、JTAG、JTAG over USB、UEFI Agent	アプリケーションおよび OS、Embedded System Registers View、MCU フラッシュ用インテルの GDB 拡張、OpenOCD* ベースの JTAG

1. インテル® System Studio は OS X* 機能の一部をサポートしています。
2. インテル® Quark™ マイクロコントローラー D1000 は LLVM ベースのインテル® コンパイラーでサポートされています。インテル® Quark™ マイクロコントローラー D2000 およびインテル® Quark™ SE マイクロコントローラーは GCC でサポートされています。
3. マイクロコントローラー向け電力アナライザーは近日提供予定です。

表 1. インテル® System Studio とインテル® System Studio for Microcontrollers の機能

インテル® System Studio

インテル® System Studio では、コマンドラインまたは GUI ベースのツール (Eclipse* または Microsoft* Visual Studio* 統合環境を含む) を選択できます。これらのツールは、ターゲットとして Windows*、Linux*、VxWorks*、Wind River Linux*、FreeBSD*、Android* をサポートしています。また、インテル® Quark™ プロセッサー、インテル® Edison プラットフォーム、インテル® Atom™ x3 プロセッサー (開発コード名 SoFIA), インテル® Atom™ x5/x7 プロセッサー (開発コード名 Cherry Trail)、第 6 世代インテル® Core™ プロセッサー (開発コード名 Skylake) を含む、最新バージョンのインテル® プロセッサーをサポートしています。

インテル® System Studio の重要なコンポーネントは、システムを解析してアーキテクチャーに対応したコードを生成する、最適化コンパイラー (インテル® C/C++ コンパイラー) と C++11 および C++14 (-std=c++14) 機能をサポートするライブラリーです。3 つのメイン・ライブラリーは、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)、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) です。

インテル® IPP は、ソフトウェア関数の広範なライブラリーにより、画像、信号、ストリング処理、データ圧縮、暗号化、コンピューター・ビジョン向けのパフォーマンス・ビルディング・ブロックを提供します。これらのライブラリーは、インテル® System Studio およびインテル® System Studio for Microcontrollers の両方でサポートされており、インテル® Quark™ プロセッサー、インテル® Atom™ プロセッサー、インテル® Core™ プロセッサー向けに追加の最適化を行います。

これらの関数はロイヤルティー・フリーで、インテル® ストリーミング SIMD 拡張 (インテル® SSE) 命令セットおよびインテル® アドバンスト・ベクトル・エクステンション (インテル® AVX、インテル® AVX2、インテル® AVX-512) 命令セットを使用して高度に最適化されているため、最適化コンパイラー単独で生成した場合よりも高速なアプリケーションを生成できます。

パフォーマンスと電力効率を最適化	インテルにより開発され、将来にわたって開発期間を短縮	広範なクロスプラットフォームと OS 機能
高度にチューニングされたルーチン	現在および以前のプロセッサー向けに最適化	高度に最適化された数千の信号、データ、メディア関数
インテル® SSE、インテル® SSSE3、インテル® SSE4、インテル® AVX、インテル® AVX2、インテル® AVX-512 命令セットを使用して高度に最適化	開発、デバッグ、保守期間を短縮	広範なドメインをサポート
最適化コンパイラー単独で生成したよりも高速	一度記述したコードで将来の最適化にも対応	インテル® Quark™ 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™ コプロセッサー・プラットフォームをサポート

表 2. インテル® System Studio の利点

インテル® MKL は、科学、工学、金融アプリケーションの演算処理を高速化します。バージョン 11.2 では、クラスター用並列直接法スパースソルバー、BLAS および LAPACK の verbose モード、小行列における S/C/Z/DGEMM の向上、SVD および固有値ソルバーのパフォーマンスの向上、このライブラリーをさらに拡張する機能と最適化が追加されました。

インテル® TBB は、タスクの並列化に広く使用されている C++ テンプレート・ライブラリーです。並列アルゴリズムとデータ構造、スレッドと同期プリミティブ、スケーラブルなメモリー割り当てとタスク・スケジュール用のテンプレートを提供します。バージョン 4.3 では、メモリー・アロケーターの改良 (改良された tbbmalloc によりマルチスレッド・アプリケーションのパフォーマンスとスケーラビリティーを向上)、インテル® トランザクショナル・シンクロナイゼーション・エクステンション (インテル® TSX) のサポートの強化 (tbb::speculative_spin_rw_mutex により読み取り/書き込みロックを使用するアプリケーションでインテル® TSX をさらに活用)、C++ 11 標準規格との互換性の向上、タスク領域の改良 (新しいクラス tbb::task_arena によりワークロードの分離と並行性レベルをより細かく制御)、最新のインテル® アーキテクチャーのサポートが行われました。(ハードウェアのサポートについては、インテル® TBB **リリースノート**を参照してください。)

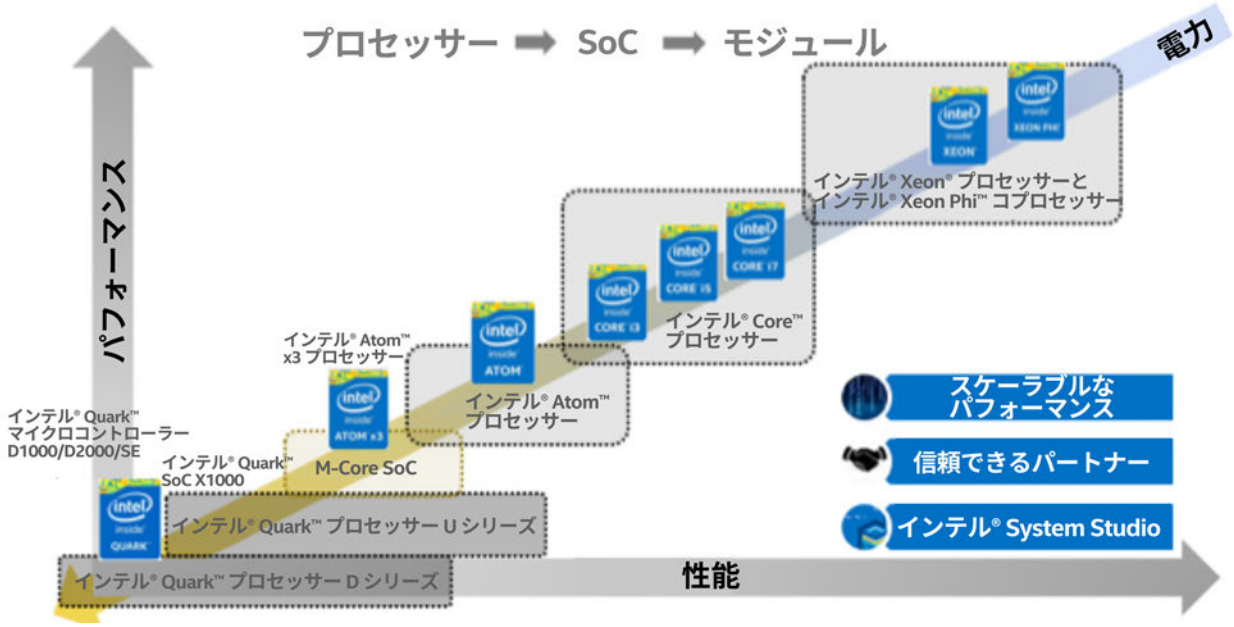
インテル® VTune™ Amplifier for Systems は、パフォーマンス・プロファイリング/電力解析ツールです。IoT ゲートウェイやモデムベースのプラットフォームでの電力解析の実行など、さまざまな解析タイプで必要なチューニング・データを迅速かつ容易に入手することができます。

インテル® System Debugger は、USB 接続による JTAG デバッグをサポートし、デバッグのニーズを満たす多くのツールを提供します。最初のツールは Microsoft* WinDbg カーネルデバッガーの JTAG デバッグおよび命令トレースです (インテル® System Studio for Windows* ターゲットバージョン)。ボード起動中の発見が困難な Windows* ドライバーの問題を特定するのを支援します。また、複雑なランタイム問題の切り分けに役立つインテル® Processor Trace (インテル® PT) を WinDbg カーネルデバッガーでサポートしています。

インテル® Trace Hub は、システム全体のハードウェアおよびソフトウェア・イベントをキャプチャーするシステムトレースを実行します。(第 6 世代インテル® Core™ プロセッサーをサポートしています。) これにより、ハードウェアとソフトウェア間の複雑な相互作用を素早く理解できます。また、タイムスタンプに関連したトレース情報を取得できます。

Closed Chassis Debug は、低コストの USB 接続による JTAG ベースのデバッグおよびトレースです。デバッグポートを利用することなく、製品要件に合ったフォームファクターで開発できます。デバッグポートの代わりに、USB ポートを利用して、デバッグに必要なすべての処理を行います。

図 1 は、ツールの IoT プロジェクトにおけるエンドツーエンドのスケーラビリティーを示しています。



1 IoT エンドツーエンドのスケーラビリティー

インテル® System Studio for Microcontrollers

インテル® System Studio for Microcontrollers は、インテル® Quark™ マイクロコントローラー D1000/D2000 およびインテル® Quark™ SE マイクロコントローラーをサポートする新しいツールスイートです。このスイートは、Eclipse* ベースの統合開発環境を提供します。開発者は、make ユーティリティーを使用して、コマンドラインからインテル® System Studio for Microcontrollers のツールを実行することもできます。

インテル® System Studio for Microcontrollers は、ベアメタルシステムや選択したリアルタイム・オペレーティング・システム (インテル® Quark™ マイクロコントローラー D2000 およびインテル® Quark™ SE マイクロコントローラー) で実行するコードの作成をサポートしています。また、ブートストラップ・コードを記述することなく、I/O 機能を単純化できるボード・サポート・パッケージ (BSP) が含まれています。

インテル® System Studio for Microcontrollers には、インテル® Quark™ マイクロコントローラー D2000 およびインテル® Quark™ SE マイクロコントローラー向け BSP に加えて、ほとんどの I/O インターフェイス (GPIO、アナログ入力、I2C、SPI、UART など) の実装とカスタマー・リファレンス・ボード (Bosch* BMC150 加速度計など) の周辺機器用のデバイスドライバーを含む、インテル® Quark™ マイクロコントローラー・ソフトウェア・インターフェイス (QMSI) パッケージが含まれています。

インテル® System Studio for Microcontrollers には、浮動小数点エミュレーションと DSP ライブラリーが含まれており、DSP ライブラリーは、パフォーマンス、精度に優れており、低消費電力で、コードサイズは高度に最適化されています (通常は関数あたり 1KB 未満)。一般的な CMSIS-DSP* ライブラリー・ベースの DSP ライブラリーには、基本的な演算、高速演算、複雑な演算、統計、変換、補間、および行列関数が含まれています。インテルが最適化した LibM には、sqrtf、expf、logf、sinf、cosf、sincosf、tanf、asinf、acosf、atanf、floorf、ceilf、truncf など、一部の使用頻度の高い単精度関数が含まれています。GNU* 標準 C 数学ライブラリーに比べて、パフォーマンスは最大で 10 倍になり、コードサイズは最小で 5 分の 1 になります。

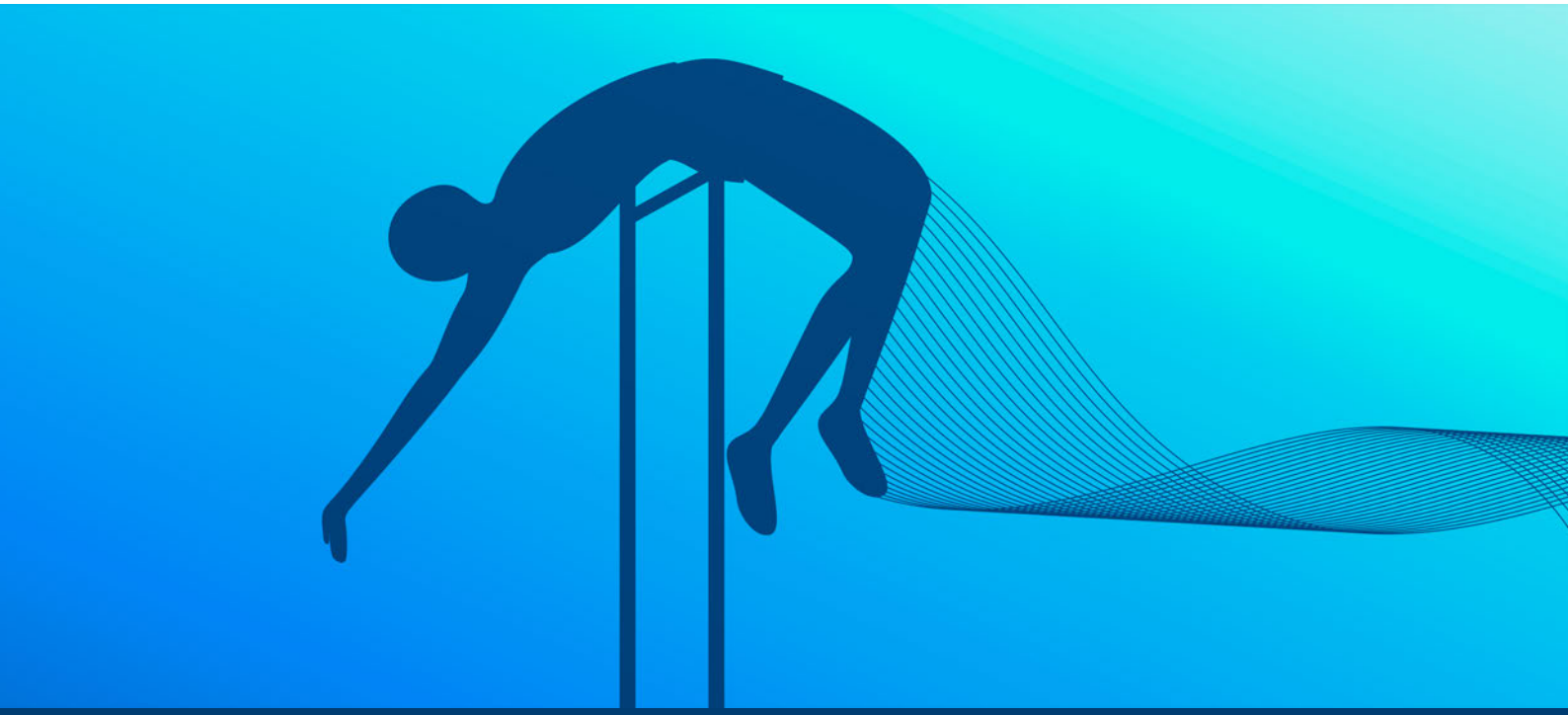
インテル® System Studio は、オープンソース GDB および OpenOCD* ソフトウェアを使用して、デバッガーとマイクロコントローラー・ファームウェア・フラッシュ・サポートを統合します。ハードウェア側の JTAG インターフェイスは、単純でコスト効率に優れた FTDI FT232H および FTDI FT2232H USB から JTAG/UART アダプターを使用して行われます。

クロスプラットフォームのツール

インテル® System Studio には、ターゲット・プラットフォームに関係なく、さまざまな組込み、モバイル、ウェアラブル、IoT、マイクロコントローラー・プラットフォームでシームレスに動作する、明確に定義されたツールのセットが用意されています。小規模なアプリケーションから、大規模なメニーコア・アプリケーションまで、これらのツールを使用することができます。この記事ではコンパイラーとライブラリーを取り上げていますが、同等のプラットフォームで動作する解析ツールも理解する価値があります。

詳細

- **インテル® System Studio**
- **インテル® System Studio for Microcontrollers**
- **インテル® MKL**
- **インテル® IPP**
- **インテル® System Debugger**
- **インテル® TBB**
- **インテル® VTune™ Amplifier for Systems**



インテルのクラスターツールを利用したハイブリッド・アプリケーションのチューニング

MPI 使用効率の理解とスレッドレベルのロードバランス

Alexey Malhanov 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
Dmitry Prohorov 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

インテル® Xeon Phi™ 製品のような現代のメニーコア・プロセッサーは、効率良いハイパフォーマンス・コンピューティング・システムの構築に必要なバランスのとれたワットあたりのパフォーマンスを実現します。しかし、コア数の増加 (メモリーサイズの増加は比較的少なくメモリー階層の複雑さは増加) は、純粋な **MPI** アプリケーションのスケーラビリティーを制限する要因となります。各 MPI ランクでのデータ複製の必要性と MPI バッファーの増加、さらに数百コアのノード内通信は、アプリケーションのパフォーマンスを制限します。MPI + X のハイブリッド・プログラミング・モデル (「X」はノード内通信に共有メモリーモデルを使用) を行うことで、ボトルネックを減らしてアプリケーションのスケーラビリティーを向上することができます。

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

最も一般的なハイブリッド・プログラミング・モデルの 1 つは MPI + OpenMP* です。これは、プラグマベースの OpenMP* はスレッドレベルの**並列化**を比較的簡単に使用でき、業界標準規格に基づいているためです。一方で、プログラミング・モデルを追加するには、MPI と「X」のスケーラビリティー問題を理解できるパフォーマンス解析ツールが必要になります。

この記事では、インテル® Parallel Studio XE Cluster Edition のコンポーネントであるインテル® VTune™ Amplifier XE とインテル® Trace Analyzer & Collector (ITAC) の機能を利用して、MPI + OpenMP* アプリケーションをチューニングします。例として、Runge-Kutta 法および有限要素法を用いて電気生理学で心臓の様子をシミュレートする、生命科学アプリケーション heart_demo を取り上げます。

ハイブリッド・アプリケーションのチューニング: どこから始めるか

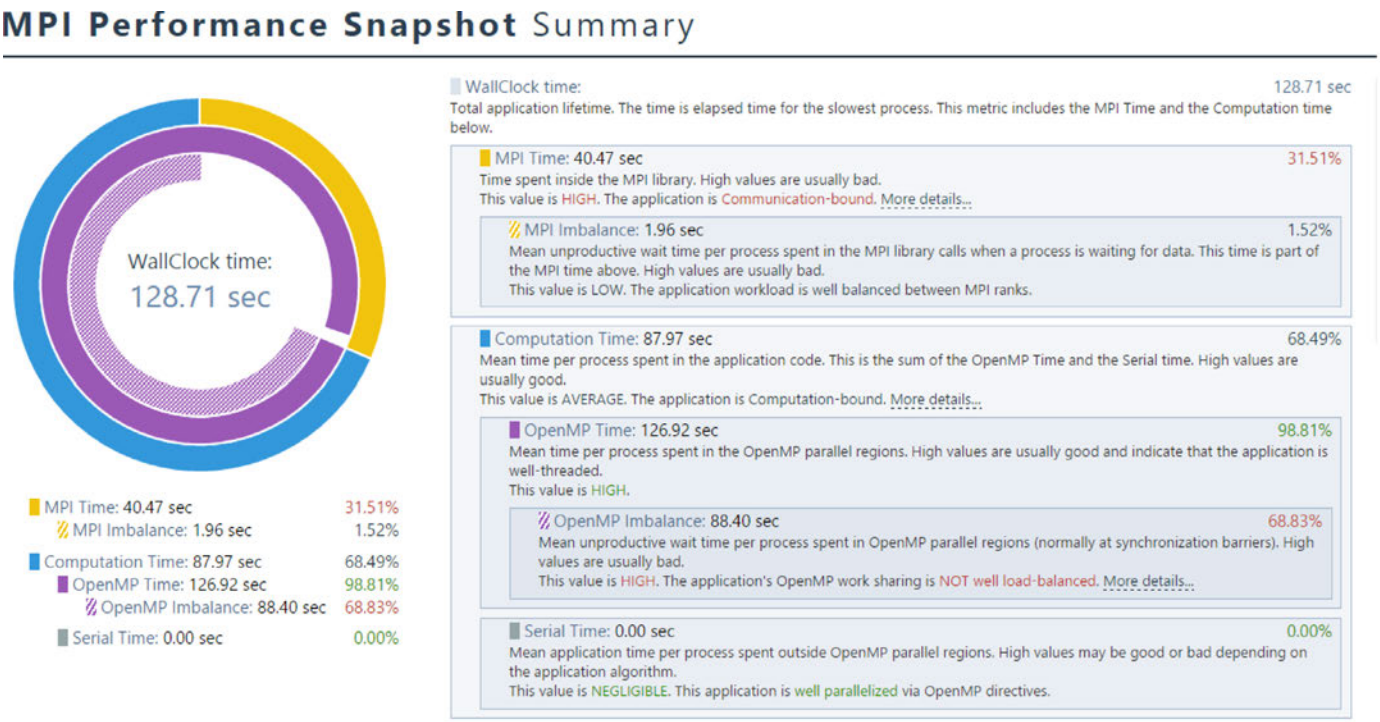
MPI と OpenMP* の両方を使用するアプリケーションを作成している開発者の目標は、ジョブに割り当てられたクラスター時間をできるだけ効率良く使用するアプリケーションを開発することです。インテル® Trace Analyzer & Collector の機能である MPI Performance Snapshot ユーティリティーは、そのためのエントリーポイントと見なすことができます。ここでは、最適なプロセス/スレッド比の識別は行わず、指定されたリソース量でアプリケーションの効率を高めることに専念します。この記事で使用しているクラスターには、2 つのソケットを持つ 4 つのノードがあり (ソケットごとに 36 CPU)、NUMA の影響を避けるためソケットごとに 1 つのプロセスを起動します。つまり、8 つの MPI プロセスと 36 の OpenMP* スレッドがあります。

MPI Performance Snapshot ユーティリティーは、アプリケーションのパフォーマンスの概要を示す軽量のユーティリティーです。MPI と OpenMP* に関連する非効率な部分を示し、アプリケーションが MPI 依存の場合は ITAC、アプリケーションに OpenMP* の利用について問題点がある場合はインテル® VTune™ Amplifier XE の解析機能を使用して詳細に調査する方法を推奨します。MPI Performance Snapshot ユーティリティーを使用した統計収集を有効にするには、**mpirun** 実行のコマンドラインに「-mps」オプションを追加します。

```
source mpivars.sh
source mpsvars.sh
mpirun -mps -n 8 -ppn 2 -hosts host1,…,host4 ./heart_demo [パラメーター]
mps stats.txt app_stat.txt -O report_initial.html
```

最初の 2 行は環境設定を行い、3 行目は MPI Performance Snapshot ユーティリティーのプロファイルを有効にしてアプリケーションを起動します。そして、最後の行はレポートを作成します。

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



1 MPI Performance Snapshot のサマリー

アプリケーションを実行した結果を**図 1** に示します。

このサマリーから、どんな結論が得られるでしょうか。まず、このアプリケーションは、MPI ライブラリーで実行時間全体の 1/3 を費やしており、効率的ではなさそうです。この時間の一部を計算に費やすことができれば、実行時間全体を減らせます。次に、MPI Performance Snapshot ユーティリティーがアプリケーションは適切にスレッド化されていると示しているにもかかわらず、OpenMP* インバランスが高くなっています。これは、ワークロードが並列領域のスレッド間で不規則に分配されていることを意味します。

ITAC を使用して MPI 依存のアプリケーションを調査する

アプリケーションが MPI 依存になる主な理由は 3 つあります。

1. MPI ライブラリー内部の待機時間が長い。これは、プロセスがほかのプロセスのデータを待っているときに発生します。この場合、MPI Imbalance インジケーターの値が高くなります (**図 1**)。
2. 通信が多い。
3. ライブラリーの最適化設定が不十分または間違って設定されている。

1 つ目と 2 つ目の項目を改善するには、通信パターンを再構築します。3 つ目の項目は、インテル® MPI ライブラリー・パッケージに含まれる mpitune ユーティリティーを使用して改善します。

では、MPI 通信を詳しく解析するため、ITAC を使用して MPI 関連の問題を調査しましょう。ITAC 内の統計収集を有効にするには、**`mpirun`** 実行のコマンドラインに **`-trace`** オプションを追加します。

```
source mpivars.sh
source itacvars.sh
mpirun -trace -n 8 -ppn 2 -hosts host1,…,host4 ./heart_demo [パラメーター]
```

ITAC をセットアップするコマンドは、最初に実行した解析で MPI Performance Snapshot ユーティリティーに使用したコマンドに似ています。3 行目のコマンドラインは、アプリケーション実行で ITAC プロファイルを有効にするように変更されています。プロファイルが完了すると、ITAC のグラフィカル・ユーザー・インターフェイスで結果を確認できます。

ITAC には、MPI の使用効率を調査するさまざまな機能が含まれています。**図 2** は、メッセージ・プロファイル・チャートです。

	P0	P1	P2	P3	P4	P5	P6	P7
P0		20.8911	17.6788	18.5337	16.941	17.3746	16.4011	17.5363
P1	59.3483		40.6097	41.5249	39.7961	39.9994	38.8246	39.9114
P2	65.7975	47.4799		48.3518	46.0318	46.2997	45.1682	46.1214
P3	60.9877	42.743	40.394		41.0926	41.4653	40.4087	41.5146
P4	62.7821	44.6009	42.3741	43.6714		44.3247	42.8955	43.7964
P5	60.698	42.5119	40.2578	41.5915	40.1263		40.7991	41.7795
P6	59.0458	40.8432	38.7668	39.9563	38.5378	38.7868		40.309
P7	55.3217	37.1146	34.9077	36.1015	34.3592	34.7522	33.3228	

2 メッセージ・プロファイル・チャート

チャートの i 番目の行と j 番目の列のセルはそれぞれ、番号 i と j のプロセス間の通信時間を示しています。チャートから、各プロセスがほかのプロセスと通信していることが分かります。プロセス 1、2、3、4、5、6、7 とプロセス 0 との間の通信時間が比較的長くなっています。これは、プロセスの 1 つ (この場合、番号 0) がほかのプロセスとの間でワークロードを分配して計算結果を収集する「マスター」プロセスの場合に特有の通信パターンです。

図 2 から、MPI の非効率な使用は、非常にアクティブな通信スキームが原因であることが分かります。この問題を解決するため、通信パターンを再考しましょう。ここでは、データを再編成して、各プロセスが前後のプロセスのみと通信を行うように通信パターンを変更しました。

	P0	P1	P2	P3	P4	P5	P6	P7
P0		24.6866						
P1	3.09586		11.6596					
P2		17.8188		11.9619				
P3			23.4029		10.891			
P4				22.6012		7.85016		
P5					25.106		7.06444	
P6						26.5887		3.77917
P7							33.8743	

3 通信パターン変更後のメッセージ・プロファイル・チャート

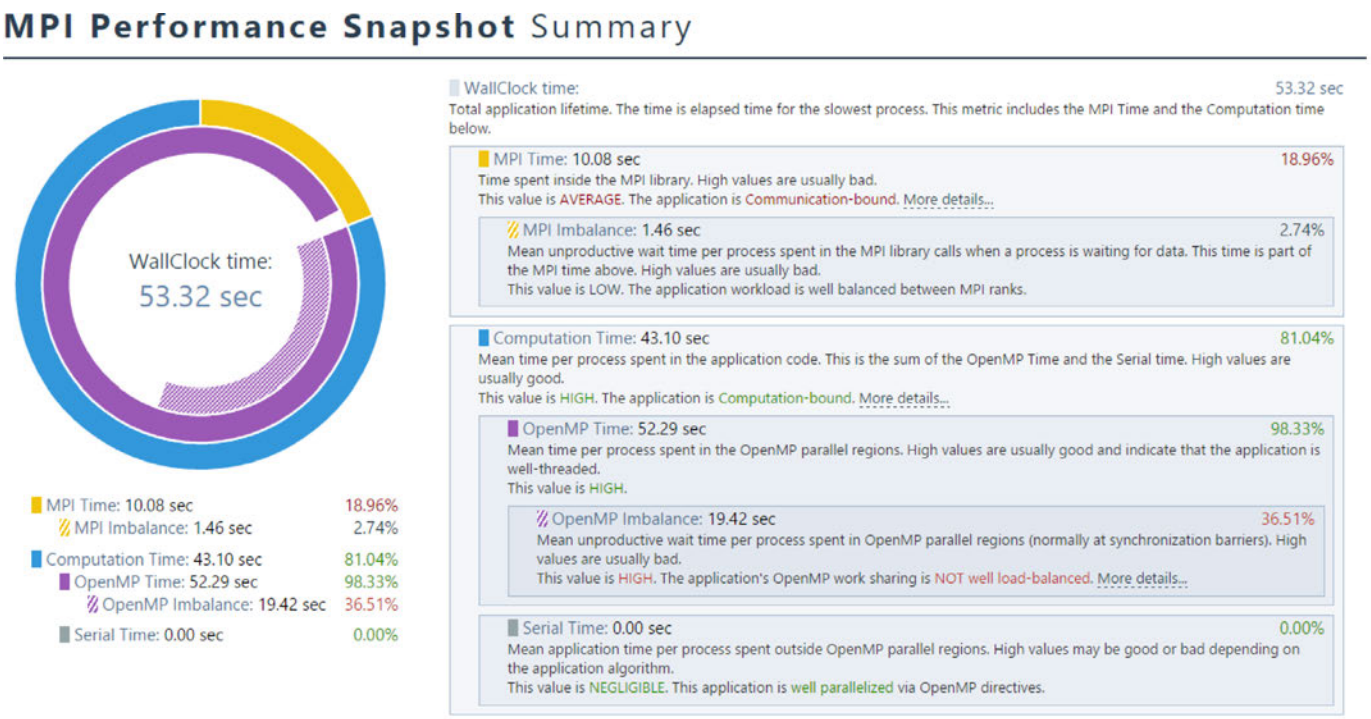
通信パターンを変更した後、メッセージ・プロファイル・チャートは図 3 のようになりました。

通信方法の変更により、プロセス間の通信が対角に行われるようになり、通信時間が大幅に短縮されました。

MPI Performance Snapshot ユーティリティーの結果は、効率が大幅に向上したことを示しています。データの再構成と通信パターンの最適化により、アプリケーションの実行時間は半分以下になりました(**図 4**)。

ただし、ツールはまだアプリケーションが MPI 依存であることを示しています。つまり、MPI の使用にまだ改善の余地があることを意味しています。しかし、目的は十分達成できているため、アプリケーションの OpenMP* 使用のチューニングに移ることにします。

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



4 効率が大幅に向上

インテル® VTune™ Amplifier XE を使用して OpenMP* インバランスを減らす

MPI Performance Snapshot ユーティリティーは、OpenMP* 並列化の効率を評価できるさまざまなメトリックを表示します。収集は、インテルの OpenMP* ランタイム・ライブラリーから生成された統計を使用します。シリアル時間 (アプリケーションが並列領域以外の実行に費やしたウォールクロック時間) とバリアーで OpenMP* スレッドが費やした時間 (バリアー同期ポイントでほかのスレッドが完了するのを待つために費やした時間) を計算する機能も含まれます。後者は、暗黙的な領域/ループまたは明示的なユーザーバリアでロード・インバランスの原因となります。ここでは、MPI Performance Snapshot ユーティリティーにより、多くの時間が OpenMP* インバランスであることを確認しました。インテル® VTune™ Amplifier XE パフォーマンス解析ツールと連携して、さらに詳細に調査することができます。

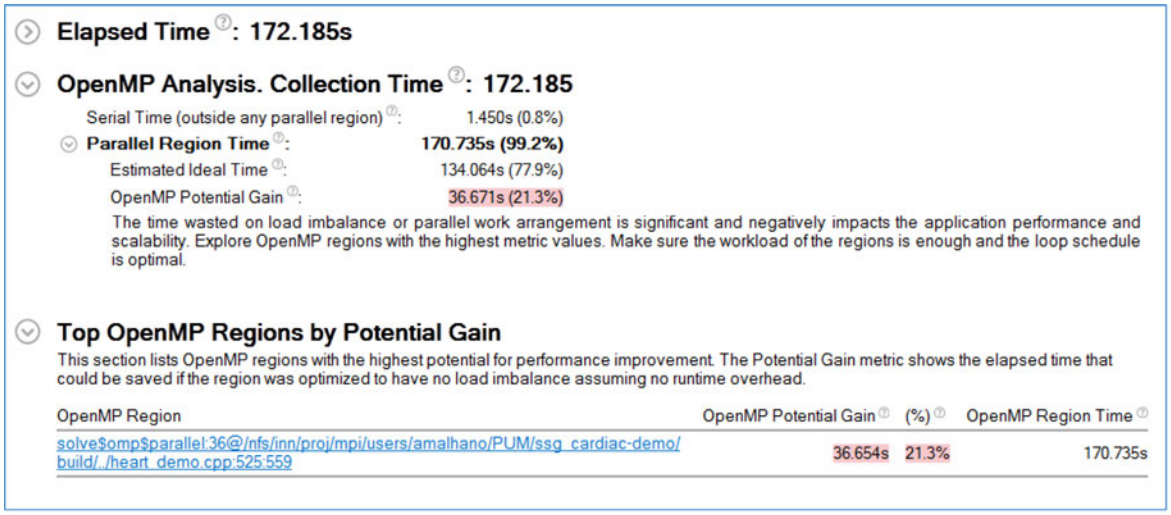
MPI Performance Snapshot ユーティリティーと同様に、インテル® VTune™ Amplifier XE はアプリケーション実行時に生成されたインテル® OpenMP* 統計を使用して次の項目を見つけます。

- OpenMP* 領域/バリアのマークアップ
- インバランスなバリア
- 並列ループの特性 (スケジュール、チャンク、ループ反復回数など)

OpenMP* ランタイムのインストルメンテーションは、アプリケーションのパフォーマンスを損なわないように注意深く使用されます。その多くは、スレッドごとにコストのかかるインストルメンテーションを回避するため、グローバルな fork-join とバリアーポイントです。スケジュール、ロック、アトミック操作、リダクション、並列処理に関連するオーバーヘッドのコストを推定するため、インテル® VTune™ Amplifier XE は、計算負荷の高いワークロードで正確な結果を生成する CPU サンプリングと統計的アプローチを使用します。

ITAC で最も CPU バウンドであると検出されたランクに Basic Hotspots 解析 (**図 5**) を使用して OpenMP* 効率を確認します。

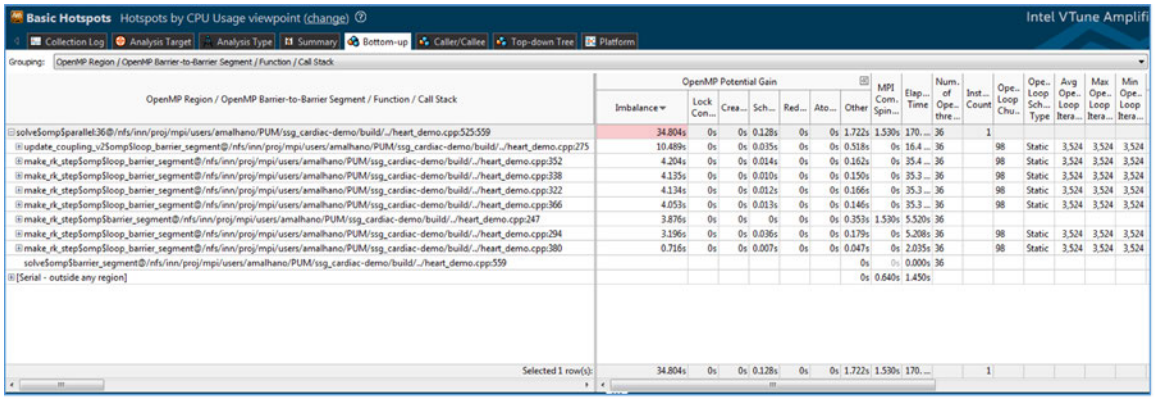
```
mpirun -gtool "amplxe-cl -collect hotspots -r result:1" -hosts
host1,…,host3 -n 8 -ppn 2 ./heart_demo [パラメーター]
```



5 Basic Hotspots 解析

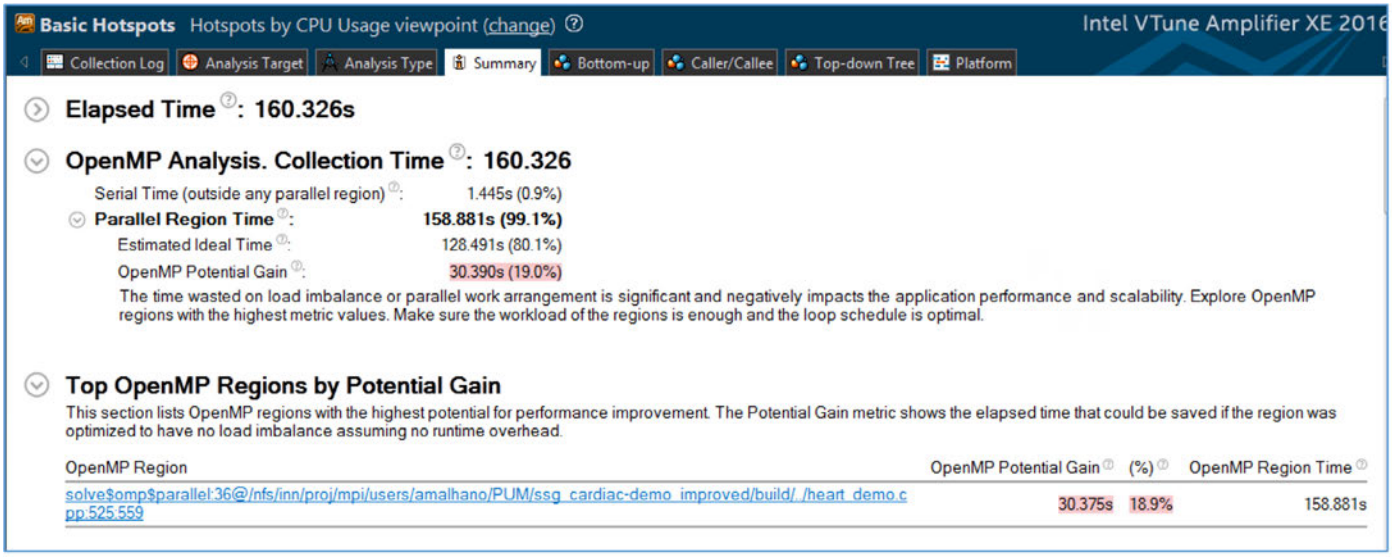
サマリービューから、アプリケーションの OpenMP* 並列コードをチューニングすることで、アプリケーションのウォールクロック時間を最大で 21% 向上できる可能性があることが分かります。これはさらに調査する価値があります。アプリケーションには 1 つの並列領域構造しかないため、Top OpenMP Regions ビューには 1 つの領域のみ含まれています。

リンクをクリックし、グリッドビューにドリルダウンしてワークシェアリング構造のバリアで領域を展開し、**図 6** の情報を表示します。



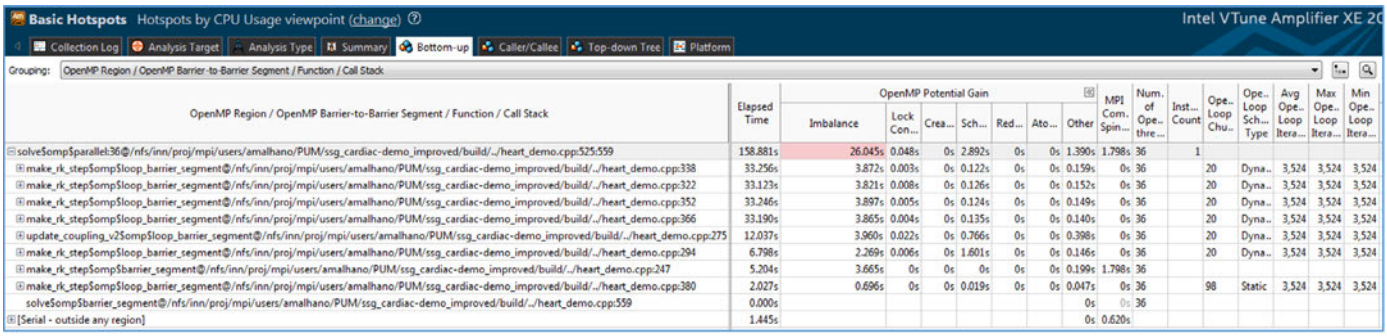
6 ワークシェアリング構造のバリアで領域を展開

Imbalance メトリックでグリッドをソートし、動的スケジュールを適用するループを選択して (リストのトップ)、インバランスを排除します。簡潔に説明するため、すべての並列ループに動的スケジュールを適用し、デフォルトでないチャンクを選択して、スケジュールのオーバーヘッド (20 反復) を排除します。動的スケジュールでプロファイルを再収集すると、結果は**図 7** のようになりました。



7 動的スケジュールでプロファイルを再収集

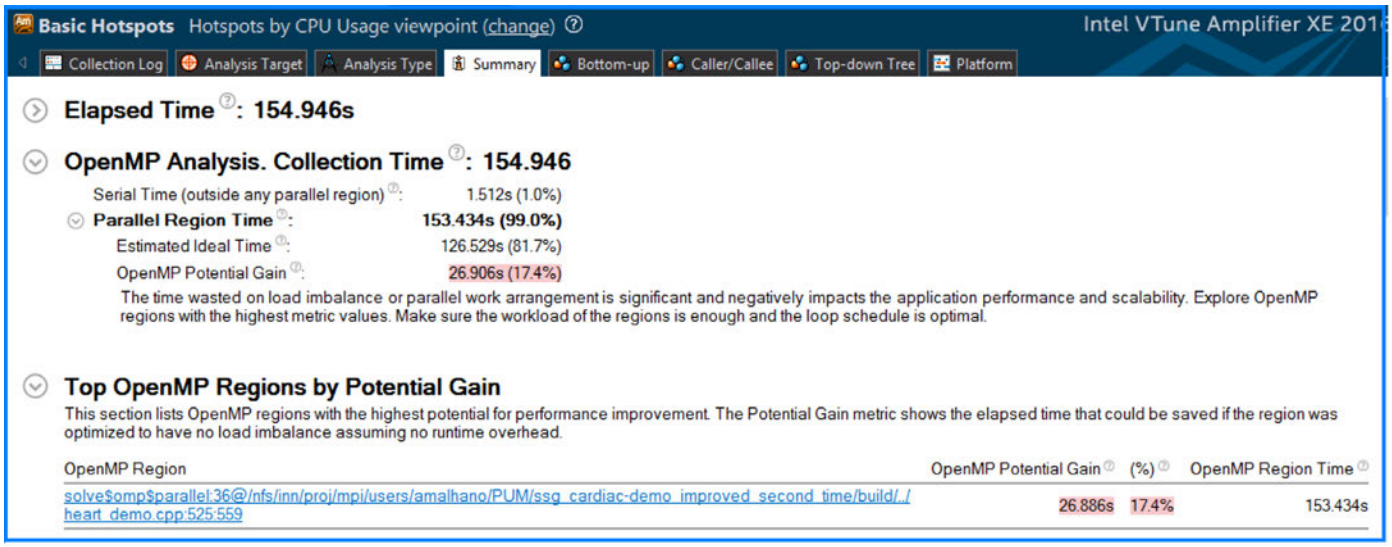
図 7 は、アプリケーションの実行時間が 12 秒短縮されたことを示しています。インバランスの時間も少なくなりました。グリッドビュー (**図 8**) は、パフォーマンスの向上に役立った主な並列ループを示しています。



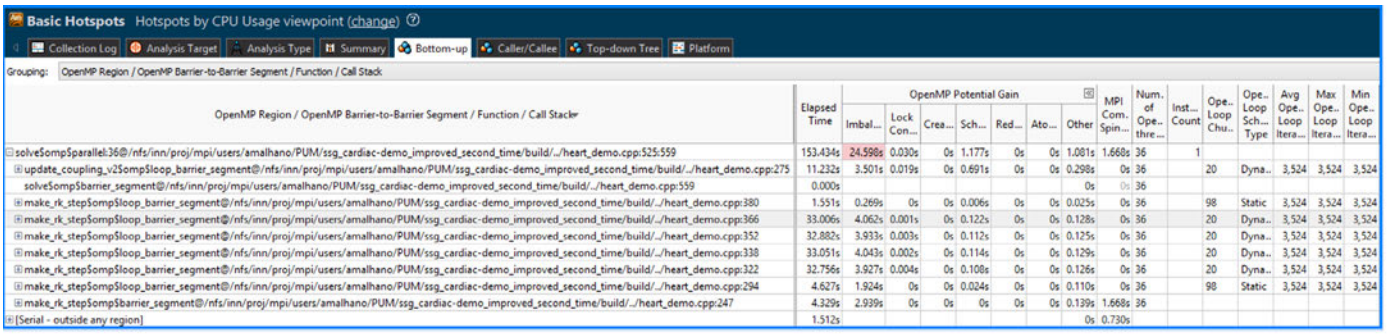
8 グリッドビュー

heart_demo.cpp の 275 行目の並列ループで、経過時間が 4 秒短くなりました。インバランスは完全に解消されていませんが、heart_demo.cpp の 322、338、352、366 行目で、ループあたり 2 秒短くなりました。スケジュールのオーバーヘッドがインバランス改善の効果を上回っているため、heart_demo.cpp の 294 行目のループは 1.5 秒長くなりました。これはスタティックにしておくほうが良いことを意味しています。この場合の結果は**図 9** と **10** のようになります。

スケジュールを行うときは、キャッシュの使用状況に注意する必要があります。ワーカースレッドによるワークの分散は動的に行われるため、キャッシュの再利用率が低下し、並列ループの時間が長くなることがあります。



9 スタティックにしてプロファイルを再収集



OpenMP Region / OpenMP Barrier-to-Barrier Segment / Function / Call Stack	Elapsed Time	OpenMP Potential Gain							MPI Com. Spin...	Num. of Ope.. thre...	Inst... Count	Ope.. Loop Chu..	Ope.. Loop Sch... Type	Avg Ope.. Loop Itera...	Max Ope.. Loop Itera...	Min Ope.. Loop Itera...
		Imbal...	Lock Con...	Crea...	Sch...	Red...	Ato...	Other								
solveompparallel:36@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo_improved_second_time/build/../heart_demo.cpp:525:559	153.434s	24.598s	0.030s	0s	1.177s	0s	0s	1.081s	1.668s	36	1					
update_coupling_v2omploop_barrier_segment@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo_improved_second_time/build/../heart_demo.cpp:275	11.232s	3.501s	0.019s	0s	0.691s	0s	0s	0.298s	0s	36		20	Dyna..	3,524	3,524	3,524
solveompbarrier_segment@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo_improved_second_time/build/../heart_demo.cpp:559	0.000s							0s	0s	36						
make_rk_stepomploop_barrier_segment@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo_improved_second_time/build/../heart_demo.cpp:380	1.551s	0.269s	0s	0s	0.006s	0s	0s	0.025s	0s	36		98	Static	3,524	3,524	3,524
make_rk_stepomploop_barrier_segment@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo_improved_second_time/build/../heart_demo.cpp:366	33.006s	4.062s	0.001s	0s	0.122s	0s	0s	0.128s	0s	36		20	Dyna..	3,524	3,524	3,524
make_rk_stepomploop_barrier_segment@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo_improved_second_time/build/../heart_demo.cpp:352	32.882s	3.933s	0.003s	0s	0.112s	0s	0s	0.125s	0s	36		20	Dyna..	3,524	3,524	3,524
make_rk_stepomploop_barrier_segment@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo_improved_second_time/build/../heart_demo.cpp:338	33.051s	4.043s	0.002s	0s	0.114s	0s	0s	0.129s	0s	36		20	Dyna..	3,524	3,524	3,524
make_rk_stepomploop_barrier_segment@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo_improved_second_time/build/../heart_demo.cpp:322	32.756s	3.927s	0.004s	0s	0.108s	0s	0s	0.126s	0s	36		20	Dyna..	3,524	3,524	3,524
make_rk_stepomploop_barrier_segment@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo_improved_second_time/build/../heart_demo.cpp:294	4.627s	1.924s	0s	0s	0.024s	0s	0s	0.110s	0s	36		98	Static	3,524	3,524	3,524
make_rk_stepompbarrier_segment@/nfs/inn/proj/mpi/users/amalhano/PUM/ssg_cardiac-demo_improved_second_time/build/../heart_demo.cpp:247	4.329s	2.939s	0s	0s	0s	0s	0s	0.139s	1.668s	36						
[Serial - outside any region]	1.512s							0s	0.730s							

10 グリッドビュー

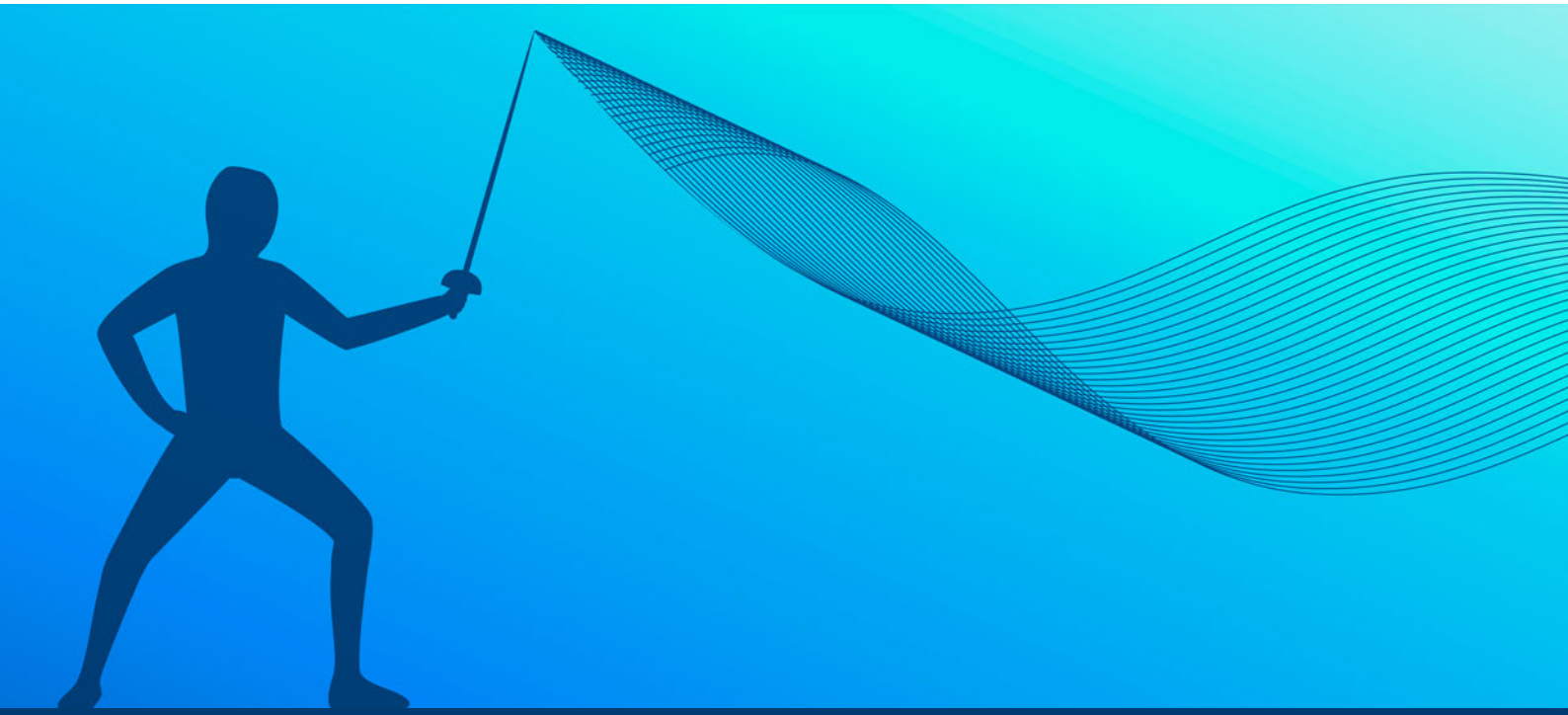
まとめ

この記事では、ハイブリッド・アプリケーションの解析とチューニングの詳細なワークフローを説明しました。ITAC のメッセージ・プロファイル・チャートは、MPI 使用効率の理解と通信パターンを変更した効率の向上に役立ちました。インテル® VTune™ Amplifier XE は、OpenMP* 並列化効率に対する考察を提供し、スレッドレベルのロードバランスの向上に役立ちました。

この記事では、ハイブリッド・アプリケーションの解析とチューニングの一部の手法のみ取り上げました。ツールには、この記事で触れていない多くの機能があります。ツールの利用シナリオはアプリケーションごとに異なります。適切なユーザーマニュアルを参照して、ツールの機能を調べ、プロジェクトにとって最適なアプローチを選択してください。

インテル® VTune™ Amplifier XE と
インテル® Trace Analyzer & Collector を評価する

インテル® Parallel Studio XE Cluster Edition に含まれます >



インテル® Advisor XE 2016 を利用したコードのベクトル化

パフォーマンスを向上する際に一般的な問題を解決する

Kevin O'Leary 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
Kirill Rogozhin インテル コーポレーション ソフトウェア開発マネージャー
Vadim Kartoshkin インテル コーポレーション テクニカルライター

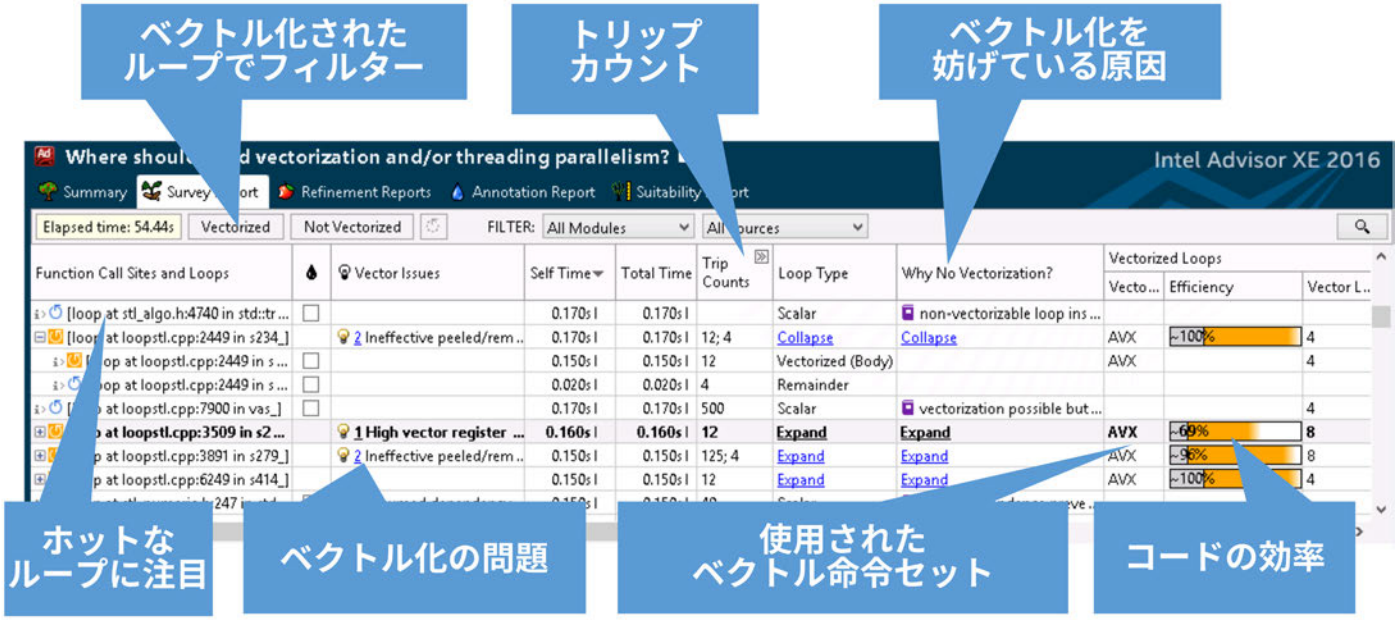
多くの要因がプログラムの自動ベクトル化を妨げます。この記事では、コンパイラーによるヒントがないとコードのベクトル化が困難である要因について検証します。ループのベクトル化は、アプリケーションのパフォーマンスを向上するために重要です。インテル® Advisor XE は、ベクトル化のプロセスをガイドします。

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

インテル® Advisor XE 2016 は、ベクトル化アドバイザーを含む動的な解析ツールです (**図 1**)。ベクトル化アドバイザーを利用することで、アプリケーションに含まれるすべてのループを調査し、次のことを確認できます。

- ベクトル化されたループとされなかったループ
- ループが**ベクトル化**されなかった原因
- ベクトル化されたループのスピードアップとベクトル化の効率
- ベクトル化の効率を下げている要因
- メモリーレイアウトにより制約を受けているベクトル化されたループとされなかったループ

この記事では、ベクトル化アドバイザーの概要を提供し、次世代のインテル® Xeon Phi™ 製品 (開発コード名 Knights Landing) 上でのベクトル化を支援する新しい機能を紹介します。また、ベクトル化アドバイザーを利用して一般的な問題をベクトル化する方法を示します。

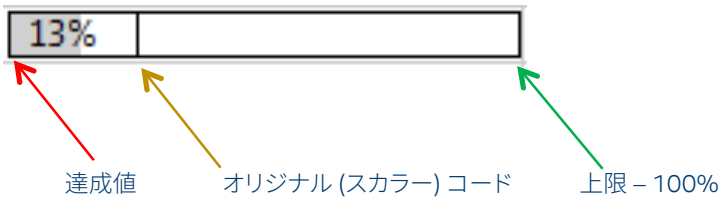


1 ベクトル化アドバイザー: 必要なすべてのデータに簡単にアクセス

ベクトル化の効率を向上するための 5 つのステップ

1. **調査。**最初のステップでは、アプリケーションを調査します。このステップで、アプリケーションが時間を費やしているループが分かります。「ホットな」ループは、最適化による恩恵が最も得られる場所です。**図 2** は、アプリケーションの [Survey Report (調査レポート)] です。インテル® Advisor XE では、ループの種類 (ベクトル化されているかどうか) でフィルターすることができます。ベクトル化されなかったループには、ベクトル化を妨げている原因が表示されます。
2. **推奨事項の確認。**ベクトル化の効率を向上する方法について具体的なアドバイスが得られます。また、ベクトル化を妨げている原因も表示されます。
3. **トリップカウント。**ループ反復のトリップカウントを個別の収集ステップとして収集します。ループがホットかどうかだけでなく、トリップカウントも把握することが重要です。トリップカウントが小さい場合、効率良くベクトル化するのに十分な反復がない可能性があります。トリップカウントがベクトル長の倍数かどうか確認することで、剰余ループが必要かどうか知ることもできます。
4. **依存性解析。**コードが正しい結果を生成するように、コンパイラーは、コンパイルしている言語のセマンティクスに対して保守的な見地に立たなければいけません。言語の規則に基づいて依存性が想定される場合、コンパイラーは依存性が存在すると仮定します。インテル® Advisor XE のような動的なツールを使用することにより、仮定した依存性が事実かどうか確認することができます。
5. **メモリー・アクセス・パターン (MAP) 解析。**データ構造がメモリー上にどのように配置され、ループでどのようにアクセスされるか知っていれば、アプリケーションのベクトル化の効率を大幅に引き上げることができます。メモリー参照がユニットストライド方式でアライメントされていることは非常に重要です。構造体配列 (AOS) から配列構造体 (SOA) へのデータ構造の変換のように、ベクトル化を支援するメモリーアクセス関連の手法があります。MAP 解析を使用すると、本質的にベクトル化が非効率なパターンを見つけ出すことができます。

Loops	Vecto...	Efficiency	Estimated Gain	Vect...	Co.	Traits	Vector Widths	Self Time
[loop at lbpSUB.cpp:1280 in fPropagationS ...	AVX	13%	0,53	4	0,53	Blends; Extracts; Inserts; Shuffles	128/256	2,312s
[loop at lbpGET.cpp:152 in fGetFracSite]	AVX	30%	2,38	8	2,34	Blends; Inserts; Masked Stores	128/256	0,030s
[loop at lbpGET.cpp:42 in fGetOneMassSite]	AVX	36%	2,86	8	2,79		256	0,100s
[loop at lbpGET.cpp:78 in fGetTotMassSite]	AVX	36%	2,86	8	2,79		256	0,010s
[loop at lbpGET.cpp:334 in fGetOneDirecSp ...	AVX	38%	3,05	8	2,97	Type Conversions	128/256	0,011s
[loop at lbpBGK.cpp:840 in fCollisionBGK]	AVX	100%	2,05	2	2,05		128	0,080s



ベクトル化の効率

2 ベクトル化の効率: パフォーマンスのサーモメーター

ベクトル化アドバイザーは、ベクトル化によるコードの効率を予測できます。効率メトリックから、対応が必要な問題を含むループが分かります。ベクトル化されたループが効率的でない場合は、最初にベクトル化アドバイザーによってコードの効率を向上するための推奨事項が提供されていないか確認します。一般に、データ構造レイアウトはベクトル化の効率に大きく影響します。MAP 解析を実行することで、ベクトル化に適した方法でメモリーを参照しているか確認できます。

インテル® AVX-512 対応ハードウェアがない場合

次世代のインテル® Xeon Phi™ コプロセッサーの実機がなくても、インテル® Advisor XE を利用して、コードを対応させることができます。インテル® コンパイラーの **-ax** オプションを使用して、複数のベクトル命令セット (インテル® AVX-512 を含む) 向けのコードを生成し、ベクトル化アドバイザーで解析します。

–ax オプションを指定してコードをコンパイルする

最初に、**-ax** オプションを指定して、コンパイラーに (デフォルトのコードパスに加え) 代替コードパスを含むバイナリーを生成するように指示します。例えば、次のコンパイラー・オプションは、インテル® SSE2 とインテル® AVX2 命令セット向けのコードを含むバイナリーを生成します。

```
-axCORE-AVX2
```

このオプションを指定すると、コンパイラーはインテル® SSE2 命令セット (**-x** または **-m** オプションで別の命令セットをデフォルトに指定しない限り、これがデフォルト) とインテル® AVX2 命令セット向けの代替コードパスを生成します。代替コードパスは、バイナリーを起動するシステムで対応する命令セットがサポートされている場合に実行されます。

ハイエンドのハードウェア (インテル® Xeon Phi™ 製品を搭載したマシンなど) 向けにコードを生成する場合は、最上位の命令セットを使用するコードの生成をコンパイラーに指示します。

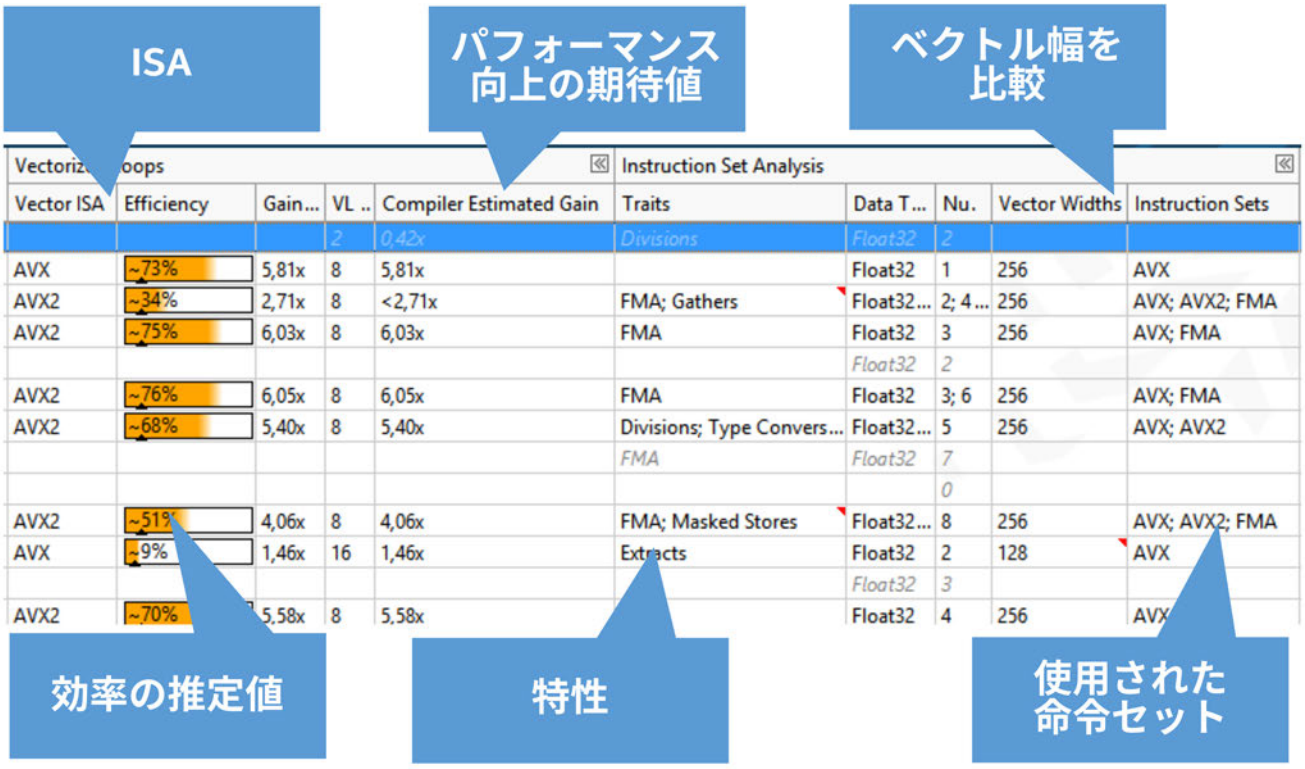
例えば、インテル® SSE4.1 以上をターゲットとし、インテル® AVX2 やインテル® AVX-512 命令セットをサポートするマシンではそれらの命令を使用するコードを生成する例について考えてみます。

この場合、コンパイラーは次のコードパスを含むコードを生成します。

```
-axCORE-AVX512, -axCORE-AVX2, -xsse4.1
```

- **–xsse4.1** は、ハードウェアが代替コードパスをサポートしていない場合、デフォルトのコードパスをインテル® SSE4.1 に変更します。
- **–axCORE-AVX2** は、インテル® AVX2 対応のハードウェア上で使用される 1 つ目の代替コードパスを設定します。
- **–axCORE-AVX512** は、インテル® AVX-512 対応のハードウェア上で使用される 2 つ目の代替コードパスを設定します。

各命令セット・アーキテクチャー (ISA) 向けに生成されたコードの違いは、インテル® Advisor XE の [Survey Report (調査レポート)] で、実行されなかったループを確認すると分かります。このレポートには、各 ISA で使用された特性に関する情報があり、パフォーマンスの期待値を比較することができます (**図 3**)。



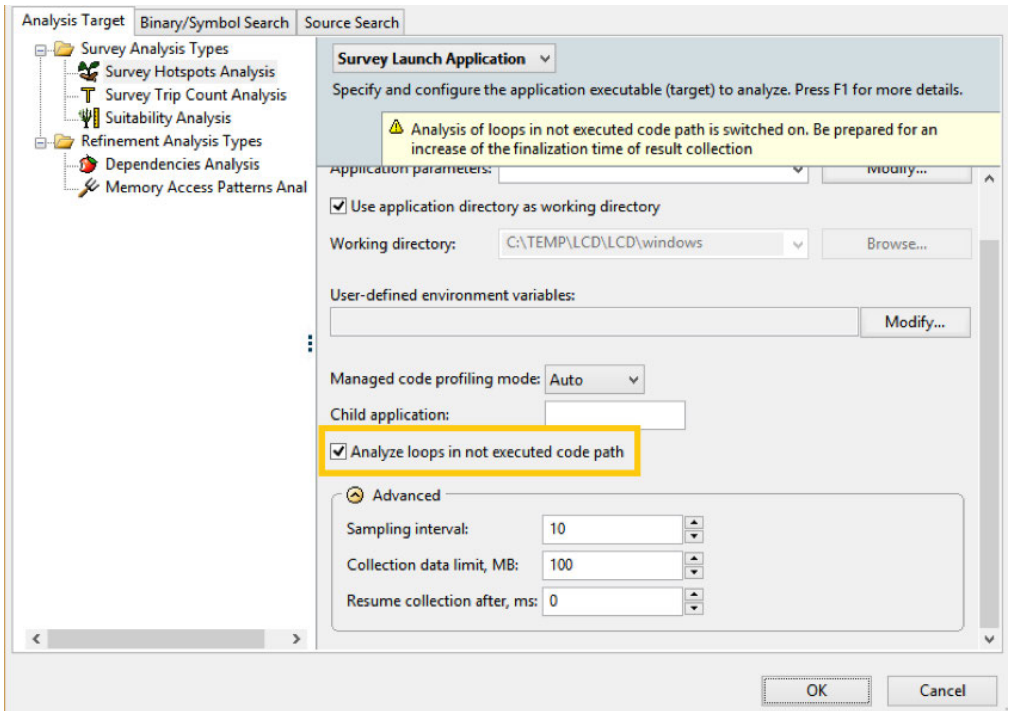
Vector ISA	Efficiency	Gain...	VL ..	Compiler Estimated Gain	Traits	Data T...	Nu.	Vector Widths	Instruction Sets
			2	0,42x	Divisions	Float32	2		
AVX	~73%	5,81x	8	5,81x		Float32	1	256	AVX
AVX2	~34%	2,71x	8	<2,71x	FMA; Gathers	Float32...	2; 4...	256	AVX; AVX2; FMA
AVX2	~75%	6,03x	8	6,03x	FMA	Float32	3	256	AVX; FMA
						Float32	2		
AVX2	~76%	6,05x	8	6,05x	FMA	Float32	3; 6	256	AVX; FMA
AVX2	~68%	5,40x	8	5,40x	Divisions; Type Convers...	Float32...	5	256	AVX; AVX2
					FMA	Float32	7		
							0		
AVX2	~51%	4,06x	8	4,06x	FMA; Masked Stores	Float32...	8	256	AVX; AVX2; FMA
AVX	~9%	1,46x	16	1,46x	Extracts	Float32	2	128	AVX
						Float32	3		
AVX2	~70%	5,58x	8	5,58x		Float32	4	256	AVX

3 インテル® Advisor XE の [Survey Report (調査レポート)]

実行されなかったコードパスの解析を有効にする

ハードウェアの対応状況に応じて複数の命令セットから最適なものを使用するバイナリーをコンパイルしたら、インテル® Advisor XE でバイナリーに含まれるすべてのバージョンのベクトルループが解析されるように、次の操作を行います。

1. インテル® Advisor XE を実行します。
2. **プロジェクトのプロパティー (Ctrl+P)** で次の操作を行います。
   a. バイナリーのパスを指定します。
   b. [Analyze loops in not executed code path (実行されなかったコードパスのループを解析する)] チェックボックスをオンにします。



3. **[OK]** をクリックします。

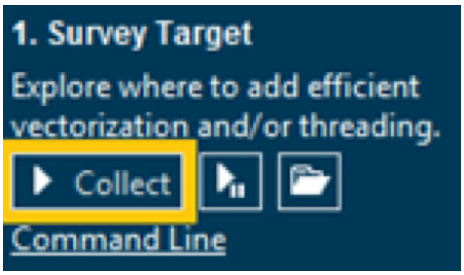
コマンドラインを使用する場合は (クラスターノード上の MPI アプリケーションなど)、次の構文を使用します。

```
mpirun -n 2 -gtool "advixe-cl -collect survey -support-multi-isa-binaries
-no-auto-finalize --project-dir=/tmp/my_proj" /tmp/bin/my_app
```

実行されなかったコードパスのループの確認

バイナリーを調査する

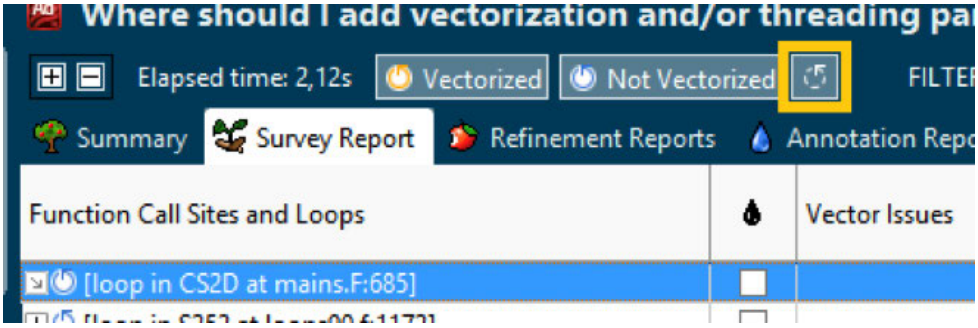
ワークフロー・タブで [Collect (収集)] ボタンをクリックします。



注: すべてのループ (現在のハードウェアで利用可能な命令セットと異なる命令セット向けの実行されなかったループを含む) の結果のファイナライズには、通常よりも長い時間がかかることがあります。

[Survey Report (調査レポート)] で実行されなかったループの表示を有効にする

調査解析結果の収集が完了したら、[Survey Report (調査レポート)] を確認します。対応するボタンをクリックして、調査グリッドで実行されなかったループの表示を有効にする必要があります。



ボタンをクリックすると、インテル® Advisor XE はグリッドをリフレッシュし、「親」ループの下に実行されなかったループを表示します。ベクトル化されたループ を展開し、実行されたループとともに実行されなかったループを確認できます (**図 4**)。

Function Call Sites and Loops	Vectorized Loops				
	Vector ISA	Efficiency	Gain...	VL (Vector Length)	Compiler Estimated Gain
[loop in S243 at loops90.f:1104]	AVX2	~76%	6,05x	8	6,05x
[loop in S243 at loops90.f:1104]	AVX2			8	6,05x
[loop in S243 at loops90.f:1104]					
[loop in S243 at loops90.f:1104]	AVX512			16	3,10x
[loop in S243 at loops90.f:1104]					
[loop in S243 at loops90.f:1104]	AVX512			32	13,29x
[loop in S243 at loops90.f:1104]	AVX512			16	5,49x
[loop in S243 at loops90.f:1104]	AVX2			4	1,49x

4 実行されたループとされなかったループ

[Survey Report (調査レポート)] で確認すべき情報

ベクトル化されたループ (Vectorized Loops) 列を確認します (ボタンで列幅を調整できます)。

インテル® Advisor XE は、実行されなかったループ に関するコンパイラーによる診断情報も表示します。

- **ベクトル ISA (Vector ISA)** 列は、特定のコードパスの ISA を示します。
- **ベクトル長 (VL (Vector Length))** 列は、ベクトルの長さを示します。この記事の執筆中に、インテル® Xeon Phi™ コプロセッサーを搭載したマシンでサンプルコードを実行したところ、ベクトルループのほうがスカラーループよりも 16 から 32 命令 (ループに応じて) 少なく、優れたパフォーマンスが得られました。
- **コンパイラーによって予測された期待値 (Compiler Estimated Gain)** 列は、コンパイラーによって予測されたパフォーマンスの期待値を示します。期待値は、ターゲット ISA をサポートするハードウェアで同じループのスカラーバージョンを実行する場合との比較です。つまり、コンパイラーは、同じインテル® AVX2 対応マシンで実行した場合、ベクトル化されたループのほうがスカラーバージョンよりも 6.05 倍パフォーマンスが向上すると予測しています。また、インテル® AVX-512 対応マシンでは、13.29 倍のパフォーマンス向上を予測しています。

命令セット解析 (Instruction Set Analysis) 列で、インテル® AVX2 とインテル® AVX-512 のループを比較することも重要です。

Call Sites and Loops	Vectorized Loops				Instruction Set Analysis				
	Vector ISA	Eff...	Gain...	VL ..	Traits	Data T...	Nu.	Vector Widths	Instructio
p in S278 at loops90.f:1526]	AVX2		5,21x	8	FMA; Masked Stores	Float32...	16	256	AVX; AVX
oop in S278 at loops90.f:1526]	AVX512			16	Unpacks; FMA; Mask Manip...	Float32...	18	512	AVX512F_
oop in S278 at loops90.f:1526]	AVX2			8	Masked Stores; FMA	Float32...	16	256	AVX; AVX
oop in S278 at loops90.f:1526]	AVX512			16	FMA	Float32...	13	512	AVX512F_
oop in S278 at loops90.f:1526]					*FMA*	*Float32...*	*3*		

- **特性 (Traits)** 列は、コードのパフォーマンスに (良くも、悪くも) 大きく影響する命令を示します。インテル® AVX-512 対応のループでは、インテル® Advisor XE はギャザー、圧縮、平方根の逆数、マスク付き操作などのインテル® AVX-512 でのみ有効な命令を特性 (Traits) として示します。
- **ベクトル幅 (Vector Width)** 列は、ベクトルレジスターの幅をビット単位で示します。この値は、ハードウェア固有です。
- **命令セット (Instruction Sets)** 列は、個々の命令で使用された命令セットを示します。

アセンブリー表現を確認する

異なる ISA (インテル® AVX-512 など) 向けコードパスのアセンブリーを表示するには、調査グリッドでループを選択し、[Loop Assembly (ループ・アセンブリー)] タブをクリックします。

Function Call Sites and Loops		Vector Issues	Self Time	Total Time	Type	Vectorized Loc / Vector ISA
[loop in S491 at loops90.f:2911]			n/a	n/a	Scalar Versions	
[loop in S233 at loops90.f:983]			0,030s	0,030s	Vectorized (Body)	AVX
[loop in S161 at loops90.f:655]		1 Poss...	0,030s	0,030s	Vectorized (Body)	AVX2
[loop in S341 at loops90.f:2254]		1 Assu..	0,010s	0,030s	Vectorized Versions	AVX2
[loop in S414 at loops90.f:2484]			**0,030s**	**0,030s**	**Vectorized (Body)**	**AVX2**
[loop in S414 at loops90.f:2484]			0,030s	0,030s	Vectorized (Body)	AVX2
[loop in S414 at loops90.f:2484]			n/a	n/a	Vectorized (Peeled) [Not Executed]	AVX512
[loop in S414 at loops90.f:2484]			n/a	n/a	Peeled [Not Executed]	
[loop in S414 at loops90.f:2484]			n/a	n/a	Vectorized (Body) [Not Executed]	AVX512
[loop in S414 at loops90.f:2484]			n/a	n/a	Vectorized (Remainder) [Not Executed]	AVX512
[loop in S414 at loops90.f:2484]			n/a	n/a	Vectorized (Remainder) [Not Executed]	AVX2
[loop in S414 at loops90.f:2484]			n/a	n/a	Remainder [Not Executed]	
[loop in VBOR at loops90.f:3349]			0,020s	0,020s	Vectorized (Body)	AVX2

Source | Top Down | Loop Analytics | Loop Assembly | Recommendations | Compiler Diagnostic Details

Module: lcd_f90.exe!0x140055c67

	Address	Line	Assembly
remainder	0x140055c67		Block 1:
	0x140055c67	2484	add r8, 0x10
	0x140055c6b	2484	vpcmpd k1, k0, zmm1, zmm0, 0x2
	0x140055c72	2484	vpaddd zmm1, k0, zmm1, zmm3
	0x140055c78	2485	vmovups zmm5, k1{z}, zmmword ptr [r15+rdx*1]
	0x140055c7f	2485	vmovups zmm6, k1{z}, zmmword ptr [r15+rbx*1]
	0x140055c86	2485	vmovups zmm4, k1{z}, zmmword ptr [r15+rbp*1]
	0x140055c8d	2485	vfmadd213ps zmm6, k0, zmm4, zmm5
	0x140055c93	2485	vmovups zmmword ptr [r15+rsi*1], k1, zmm6
	0x140055c9a	2484	add r15, 0x40
	0x140055c9e	2484	cmp r8, rax
	0x140055ca1	2484	jb 0x140055c67 <Block 1>

この機能を利用して、インテル® AVX2 とインテル® AVX-512 のコードパスの違いを確認できます。

ループのベクトル化に関する予備知識

一般的にベクトル化されたループは次の 3 つの部分で構成されます。

- **メインのベクトル本体:** 3 つの部分の中で最も高速です。
- **オプションのピール部分:** ループのアライメントされていないメモリー参照に使用されます。スカラーか、低速なベクトル命令を使用します。
- **剰余部分:** 反復回数 (トリップカウント) がベクトル長の倍数でない場合に生成されます。スカラーか、低速なベクトル命令を使用します。

ベクトルレジスター幅が長くなると、ループのピール/剰余部分の反復回数が多くなります。

- データはアライメントし、コンパイラーにアライメントされていることを伝えます。
- 反復回数は、ベクトル長の倍数になるようにします。

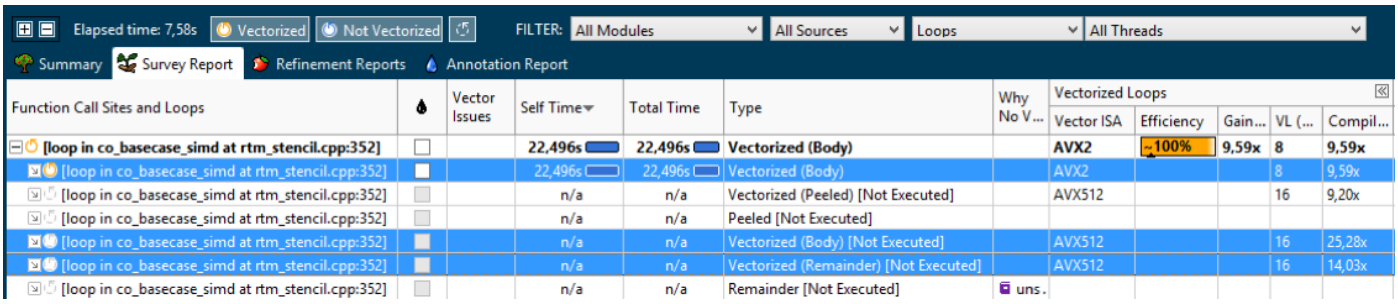
インテル® AVX-512 の診断例

RTM ステンシル・プロジェクト

ステンシル計算は、地震探査の RTM (Reverse Time Migration: リバース・タイム・マイグレーション) アルゴリズムの基本です。有限差分法により波動方程式を解きます。このサンプルは、3 次元の 25 ポイントステンシルです。**RTM ステンシル**サンプル用にインテル® AVX-512 コードを生成したところ、コンパイラーはインテル® AVX-512 では 25.28 倍、インテル® AVX2 では 9.59 倍のスピードアップを予測しました (**図 5**)。

ベクトル長が 2 倍のため、インテル® AVX-512 コードは 2 倍高速になると予測しましたが、コンパイラーによって示された期待値は 2.63 倍でした。

その理由は、インテル® Advisor XE により解明できます。インテル® AVX2 コードにはスカラーの剰余部分がありますが、この剰余部分がインテル® AVX-512 ではベクトル化されています。



Function Call Sites and Loops		Vector Issues	Self Time	Total Time	Type	Why No V...	Vectorized Loops				
							Vector ISA	Efficiency	Gain...	VL (...	Compil...
[loop in co_basecase_simd at rtm_stencil.cpp:352]			22,496s	22,496s	Vectorized (Body)		AVX2	~100%	9,59x	8	9,59x
[loop in co_basecase_simd at rtm_stencil.cpp:352]			22,496s	22,496s	Vectorized (Body)		AVX2			8	9,59x
[loop in co_basecase_simd at rtm_stencil.cpp:352]			n/a	n/a	Vectorized (Peeled) [Not Executed]		AVX512			16	9,20x
[loop in co_basecase_simd at rtm_stencil.cpp:352]			n/a	n/a	Peeled [Not Executed]						
[loop in co_basecase_simd at rtm_stencil.cpp:352]			n/a	n/a	Vectorized (Body) [Not Executed]		AVX512			16	25,28x
[loop in co_basecase_simd at rtm_stencil.cpp:352]			n/a	n/a	Vectorized (Remainder) [Not Executed]		AVX512			16	14,03x
[loop in co_basecase_simd at rtm_stencil.cpp:352]			n/a	n/a	Remainder [Not Executed]	uns.					

5 RTM ステンシルサンプルのスピードアップ

LCD ベクトル化ベンチマーク

このサンプルでは、コンパイラーは 2 つのバージョンの期待値を 12.20 倍 (インテル® AVX2) と 36.34 倍 (インテル® AVX-512) と予測しました。インテル® AVX-512 コードのほうが 2.97 倍高速ですが、ベクトル長は 2 倍です。RTM ステンシルサンプルでは、コンパイラーはインテル® AVX2 の剰余ループをベクトル化できませんでした。

このサンプルでは、インテル® AVX2 とインテル® AVX-512 の剰余部分はともにベクトル化されています。期待値が 2 倍を上回った理由は、インテル® AVX-512 のマスク付き操作によるものです。

インテル® AVX2:

Type	Why No Vectori...	Vectorized Loops					Instruction Set Analysis					Advanced		
		Vector ISA	Efficien...	Gain...	VL (...	Compiler Esti...	Traits	Data Types	Number of Ve...	Vector Widths	Instruction Sets	Transfor...	Unroll Fa...	Vectorization Details
Vectorized (Body) [Not Executed]		AVX2			8	12,20x	FMA	Float32	4	256	AVX; FMA	Unrolled	2	Unaligned Access in Vector Loop
Vectorized (Remainder) [Not Executed]		AVX512			16	16,88x	FMA	Float32; Int32...	6	512	AVX512F_512			Masked Loop Vectorization; Unaligned Access
Vectorized (Remainder) [Not Executed]		AVX512			16	16,88x	FMA	Float32; Int32...	6	512	AVX512F_512			Masked Loop Vectorization; Unaligned Access
Vectorized (Remainder) [Not Executed]		AVX2			4	2,68x	FMA	Float32	2	128	AVX; FMA			Unaligned Access in Vector Loop
Vectorized (Remainder) [Not Executed]		AVX2			4	2,68x	FMA	Float32	2	128	AVX; FMA			Unaligned Access in Vector Loop
Vectorized (Body) [Not Executed]		AVX512			32	36,34x	FMA	Float32	4	512	AVX512F_512	Unrolled	1	Unaligned Access in Vector Loop

Source | Top Down | Loop Analytics | Loop Assembly | Recommendations | Compiler Diagnostic Details

Module: lcd_cxxi.exe!0x14005c4b3

	Address	Line	Assembly	Total Time	%	Self Time	%	Traits
remainder	0x14005c4b3		Block 1:					
	0x14005c4b3	6252	vmovups xmm1, xmmword ptr [rcx+rsi*4]					
	0x14005c4b8	6252	vmovups xmm0, xmmword ptr [rdi+rsi*4]					
	0x14005c4bd	6252	vfmadd213ps xmm1, xmm0, xmmword ptr [r8+rsi*4]					FMA
	0x14005c4c3	6252	vmovups xmmword ptr [r8+rsi*4], xmm1					
	0x14005c4c9	6251	add rsi, 0x4					
	0x14005c4cd	6251	cmp rsi, r12					
	0x14005c4d0	6251	jb 0x14005c4b3 <Block 1>					

インテル® AVX-512:

Type	Why No Vectori...	Vectorized Loops					Instruction Set Analysis					Advanced		
		Vector ISA	Efficiency	Gain...	VL (...	Compiler Estimated ...	Traits	Data Types	Number of Ve...	Vector Widths	Instruction Sets	Transfor...	Unr.	Vectorization Details
Vectorized (Body) [Not Executed]		AVX2			8	12,20x	FMA	Float32	4	256	AVX; FMA	Unrolled	2	Unaligned Access in Vector Loop
Vectorized (Remainder) [Not Executed]		AVX512			16	16,88x	FMA	Float32; Int32...	6	512	AVX512F_512			Masked Loop Vectorization; Unaligned Access
Vectorized (Remainder) [Not Executed]		AVX512			16	16,88x	FMA	Float32; Int32...	6	512	AVX512F_512			Masked Loop Vectorization; Unaligned Access
Vectorized (Remainder) [Not Executed]		AVX2			4	2,68x	FMA	Float32	2	128	AVX; FMA			Unaligned Access in Vector Loop
Vectorized (Remainder) [Not Executed]		AVX2			4	2,68x	FMA	Float32	2	128	AVX; FMA			Unaligned Access in Vector Loop
Vectorized (Body) [Not Executed]		AVX512			32	36,34x	FMA	Float32	4	512	AVX512F_512	Unrolled	1	Unaligned Access in Vector Loop

Source | Top Down | Loop Analytics | Loop Assembly | Recommendations | Compiler Diagnostic Details

Module: lcd_cxxi.exe!0x14005bea8

	Address	Line	Assembly	Total Time	%	Self Time	%	Traits
remainder	0x14005bea8		Block 1:					
	0x14005bea8	6251	add rsi, 0x10					
	0x14005beac	6251	vpcmpd k1, k0, zmm3, zmm2, 0x2					
	0x14005beb3	6251	vpaddd zmm3, k0, zmm3, zmm1					
	0x14005beb9	6252	vmovups zmm5, k1{z}, zmmword ptr [r8+r15*1]					
	0x14005bec0	6252	vmovups zmm10, k1{z}, zmmword ptr [r8+rbp*1]					
	0x14005bec7	6252	vmovups zmm4, k1{z}, zmmword ptr [r8+rbx*1]					
	0x14005bece	6252	vfmadd213ps zmm10, k0, zmm4, zmm5					FMA
	0x14005bed4	6252	vmovups zmmword ptr [r8+rcx*1], k1, zmm10					
	0x14005bedb	6251	add r8, 0x40					
	0x14005bedf	6251	cmp rsi, rax					
	0x14005bee2	6251	jb 0x14005bea8 <Block 1>					

ベクトル化の詳細情報から、インテル® AVX-512 コードでは剰余ループでマスク付き操作が使用されており、16.88 倍のスピードアップにつながっています。一方、インテル® AVX2 の剰余ループは 2.60 倍のスピードアップにとどまっています。

マスク付き操作を利用して、インテル® AVX-512 バージョンでは「ピール」ループもベクトル化しています。次のループは、インテル® AVX-512 バージョンではピール、本体、剰余ループがベクトル化されます。

Function Call Sites and Loops	Vect... Issues	Self Tim.	Total Time	Type	Why No Vectori...	Vectorized Loops: Vector ISA	Efficien...	Gain E...	VL (...	Compiler Es...	Instruction Set Analysis: Traits	Data Types	Number of Ve...	Vector Widths	Instruction Sets
[loop in s234_ at loopstl.cpp:2449]		**0,030s**	**0,030s**	**Vectorized (Body)**		**AVX2**	**100%**	**11,52x**	**8**	**11,52x**	**FMA**	**Float32**	**4**	**256**	**AVX; FMA**
[loop in s234_ at loopstl.cpp:2449]		0,030s	0,030s	Vectorized (Body)		AVX2			8	11,52x	FMA	Float32	4	256	AVX; FMA
[loop in s234_ at loopstl.cpp:2449]		n/a	n/a	Peeled [Not Executed]							FMA	Float32	2		
[loop in s234_ at loopstl.cpp:2449]		n/a	n/a	Peeled [Not Executed]							FMA	Float32	2		
[loop in s234_ at loopstl.cpp:2449]		n/a	n/a	Scalar [Not Executed]	non ...						FMA	Float32	4		
[loop in s234_ at loopstl.cpp:2449]		n/a	n/a	Scalar [Not Executed]	non ...						FMA	Float32	4		
[loop in s234_ at loopstl.cpp:2449]		n/a	n/a	Remainder [Not Executed]							FMA	Float32	2		
[loop in s234_ at loopstl.cpp:2449]		n/a	n/a	Remainder [Not Executed]							FMA	Float32	2		
[loop in s234_ at loopstl.cpp:2449]		n/a	n/a	Vectorized (Peeled) [Not Executed]		AVX512			16	9,00x	FMA	Float32; Int32...	6	512	AVX512F_512
[loop in s234_ at loopstl.cpp:2449]		n/a	n/a	Vectorized (Peeled) [Not Executed]		AVX512			16	9,00x	FMA	Float32; Int32...	6	512	AVX512F_512
[loop in s234_ at loopstl.cpp:2449]		n/a	n/a	Vectorized (Remainder) [Not Executed]		AVX512			16	16,88x	FMA	Float32; Int32...	6	512	AVX512F_512
[loop in s234_ at loopstl.cpp:2449]		n/a	n/a	Vectorized (Remainder) [Not Executed]		AVX512			16	16,88x	FMA	Float32; Int32...	6	512	AVX512F_512
[loop in s234_ at loopstl.cpp:2449]		n/a	n/a	Vectorized (Body) [Not Executed]		AVX512			32	38,97x	FMA	Float32	4	512	AVX512F_512
[loop in s234_ at loopstl.cpp:2449]		n/a	n/a	Vectorized (Body) [Not Executed]		AVX512			32	36,34x	FMA	Float32	4	512	AVX512F_512
[loop in s234_ at loopstl.cpp:2449]		n/a	n/a	Vectorized (Remainder) [Not Executed]		AVX2			4	2,68x	FMA	Float32	2	128	AVX; FMA
[loop in s234_ at loopstl.cpp:2449]		n/a	n/a	Vectorized (Remainder) [Not Executed]		AVX2			4	2,68x	FMA	Float32	2	128	AVX; FMA
[loop in s234_ at loopstl.cpp:2449]		n/a	n/a	Vectorized (Body) [Not Executed]		AVX2			8	12,20x	FMA	Float32	4	256	AVX; FMA

インテル® AVX-512 のベクトル化された剰余ループは、ベクトル長 16 をすべて使用しているのに対し、インテル® AVX2 バージョンはベクトル長 4 のみ—インテル® AVX2 の本体よりも少ない—を使用しています。インテル® AVX2 バージョンのピールループはスカラーのみであるのに対し、インテル® AVX-512 のピールループはベクトル化され、9.0 倍のスピードアップが期待できます。

要点のおさらい

インテル® Advisor XE では、次のことが可能です。

- 一度に、1 台のマシンで、複数の ISA 向けのコードを生成し、解析することができます。また、コンパイラー・レポートからパフォーマンスの期待値が得られます。
- 個々の命令で使用された ISA 固有の命令の「ファミリー」や、コードパス固有の特性を確認できます。

関連情報 (英語):

- **インテル® C++ コンパイラーのコード生成オプション**
- **インテル® Fortran コンパイラーのコード生成オプション**
- **インテル® C++ コンパイラーの x、Qx オプション**
- **インテル® Fortran コンパイラーの x、Qx オプション**
- **インテル® C++ コンパイラーの ax、Qax オプション**
- **インテル® Fortran コンパイラーの ax、Qax オプション**

まとめ

この記事では、インテル® Advisor XE 2016 の概要と、インテル® AVX-512 ISA 向けのコードをインテル® AVX2 対応プロセッサー・ベースのマシンで実行し、解析する機能を説明しました。また、サンプルを用いてベクトル化アドバイザーにより C++ STL コードをベクトル化する方法を示しました。

ハードウェアを最大限に活用するには、ベクトル化とスレッド化を導入してコードを現代化する必要があります。この記事で説明した系統的なアプローチを採用し、インテル® Parallel Studio XE の強力なツールを活用することで、簡単にこの作業を行うことができます。

インテル® Advisor XE を評価する
インテル® Parallel Studio XE に含まれます >

Software

THE PARALLEL UNIVERSE

© 2016 Intel Corporation. 無断での引用、転載を禁じます。Intel、インテル、Intel ロゴ、Cilk、Intel Atom、Intel Core、Intel Xeon Phi、Pentium、Quark、VTune、Xeon は、アメリカ合衆国および / またはその他の国における Intel Corporation の商標です。* その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

OpenCL および OpenCL ロゴは、Apple Inc. の商標であり、Khronos の使用許諾を受けて使用しています。