THE PARALLEL UNIVERSE

Issue 10
2012 年 6 月

6 ステップの ベクトル化

将来も通用するアプリケーションの生成

Shannon Cepeda、Wendy Doerner

intel Software

インテル® Cilk™ Plus
パフォーマンス・ブースター

インテル® Cilk™ Plus がオープンソースになり GCC* 4.7 に対応しました。

インテル® Cilk™ Plus の 3 つのキーワードは、簡単で強力な並列プログラミング・モデルを提供します。

- ランタイム・ライブラリーとテンプレート・ライブラリーにより、並列アプリケーションのビルド用に適切にチューニングされた環境を提供します。
- Linux*、Windows*、オープンソース、GCC* 4.7 の C++/Fortran 開発者に最適です。

インテル® Parallel Studio XE の詳細

Rock your code. **Rock your world.**

©2012 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、Pentium、Xeon、VTune は、アメリカ合衆国および / またはその他の国における Intel Corporation の商標です。* その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

目次

編集者からのメッセージ

パフォーマンス向上への取り組み:
具体的な実現方法 - クラスターからデバイスまで
JAMES REINDERS 4

6 ステップのベクトル化による将来も通用するアプリケーションの生成

WENDY DOERNER、SHANNON CEPEDA 6

ベクトル化は、データ並列化をサポートする重要なハードウェア機能で、単一プロセッサー・コア内で並列にデータを処理できるようにし、パフォーマンスを大幅に向上させます。スレッド化やクラスター並列化と組み合わせることもできます。

さまざまな環境に応じた並列計算

ROBERT MUELLER、NOAH CLEMONS、PAUL FISCHER 12

スモール・フォーム・ファクター・デバイスで動作するアプリケーションにパフォーマンス・ライブラリー関数を利用することは、データ集約型タスクの計算実行フローを合理化して統一する優れた方法です。この記事では、インテル® Atom™ プロセッサーでインテル® 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) を活用する方法を検討します。

.NET アプリケーションの信頼性とパフォーマンスを高めるツール

LEVENT AKYIL、ASAF YAFFE 20

.NET コード、ネイティブコード、「混合」(.NET およびネイティブ) アプリケーションの開発サイクルで開発者を支援する最新のソフトウェア・ツールの機能を紹介します。

© 2012 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、Pentium、Xeon、VTune は、アメリカ合衆国および / またはその他の国における Intel Corporation の商標です。* その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

パフォーマンス向上への取り組み：

具体的な実現方法

～クラスターからデバイスまで～

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

編集者からのメッセージ

James Reinders は、インテル コーポレーションのチーフ・エバンジェリストです。『Intel Threading Building Blocks: Outfitting C++ for Multicore Processor Parallelism』（日本語:『インテル スレッディング・ビルディング・ブロック ーマルチコア時代の 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）などの並列化に関する文献を発表しています。また、並列化に関する幅広いインタビューも受けています。

アプリケーションのパフォーマンスを向上させるテクニックとヒントがさまざまなソフトウェア技術者から寄せられたこの号も、開発者に必見の情報が満載されています。優れた結果を得るために複雑にする必要はありません。基本的な並列化プロセスの多くは、インテル® ソフトウェア開発ツールを利用して効率良く行うことができます。

「6 ステップのベクトル化による将来も通用するアプリケーションの生成」では、複数のデータに対して特定の処理を一度に実行する、ベクトル並列化と呼ばれる並列化手法を紹介します。スケールフォワード手法を用いることで、新しいアーキテクチャーやプロセッサーに合わせてコードを変更することなく、コードをベクトル化することができます。

「さまざまな環境に応じた並列計算:インテル® Atom™ プロセッサーでインテル® 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）を活用する」では、モバイル・アプリケーションの開発者にとって興味深い手法を用いて、スモール・フォーム・ファクター・デバイスに並列化を適用する方法を紹介します。

そして、「.NET アプリケーションの信頼性とパフォーマンスを高めるツール」では、インテル® Parallel Studio XE に含まれる 2 つのツール（エラー検出ツール インテル® Inspector XE とパフォーマンス分析ツール インテル® VTune™ Amplifier XE）が .NET コード、ネイティブコード、「混合」（.NET およびネイティブ）アプリケーションの開発サイクルにおいてどのように開発者に役立つかを説明します。

この号で紹介するコードを実際に試し、開発プロセスとソフトウェアに与える影響を検証して、皆さんがパフォーマンス・ツールキットの幅広い活用方法を習得し、パフォーマンスを大幅に向上されることを願っています。

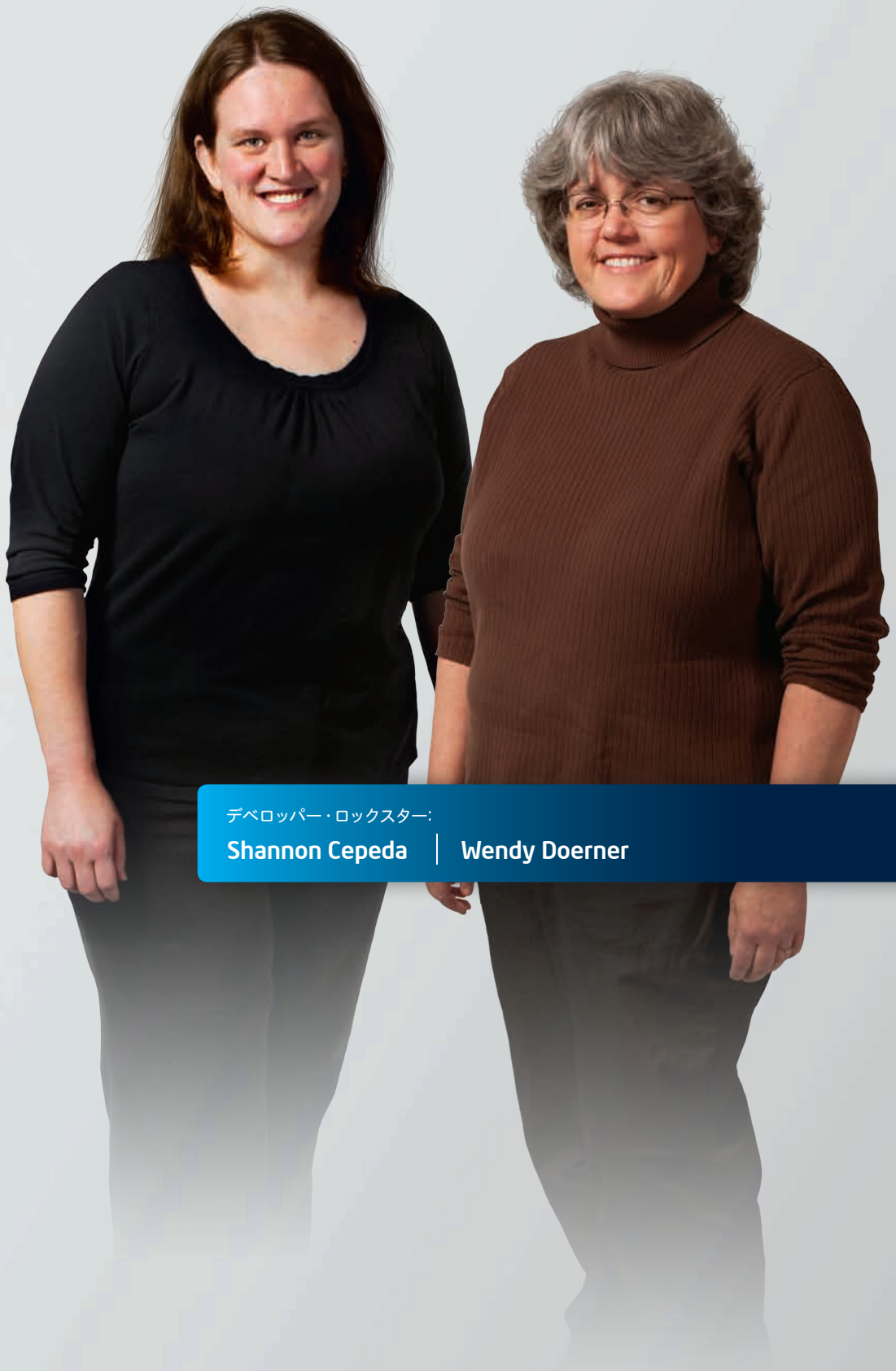
James Reinders

6 ステップの ベクトル化

将来も通用する アプリケーションの生成

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
Wendy Doerner、Shannon Cepeda

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。



デベロッパー・ロックスター:
Shannon Cepeda | Wendy Doerner

単一プロセッサー・コアでデータを並列に処理することでパフォーマンスを大幅に向上させる方法を紹介します。そして、スレッド化とクラスター並列化を組み合わせて、さらなるパフォーマンス向上を図ります。

Parallel Universe ではこれまで、スレッド化やクラスター・プログラミングによって複数のコアを活用することを目標とした、並列化に関する多くの記事を掲載してきました。この記事では、ベクトル並列化と呼ばれる重要な並列化手法を紹介します。ベクトル化は、データ並列化をサポートするハードウェア機能で、単一プロセッサー・コア内で並列にデータを処理します。

ベクトル化によって、複数のデータに対して特定の処理を一度に実行することができます。この処理は、SIMD (Single Instruction Multiple Data) と呼ばれる特別な命令を使用して行われます。SIMD 命令と SIMD 命令に対応するインテル® プロセッサーは、10 年以上にわたり進化し続けてきました。インテル独自の SIMD 命令セットであるインテル® ストリーミング SIMD 拡張命令 (インテル® SSE) は、1999 年に最初に導入された後、数回の拡張が行われています。2011 年には、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) が登場しました。

ベクトル化はどのように動作するか

典型的なスカラー (非ベクトル) の場合、使用する変数はそれぞれ、独自のプロセッサー・レジスターに別々に格納されます。2 つの変数で加算を実行すると、2 つのレジスターの内容が加算され、結果がレジスターに戻されます。この例のベクトル化では、最初に加算対象の複数の変数をレジスターに「パック」します (埋めます)。例えば、インテル® AVX 対応のプロセッサーでは、最大 8 つの単精度浮動小数データ要素を 1 つのレジスターにパックできます。そして、1 つの SIMD 命令により、これらのデータ要素と別の要素がパックされたレジスターを使って操作し、一度に複数の結果を生成します。これらの操作は別々ではなく並行に行えることから、適切なコードでは大幅なパフォーマンス向上を達成できます。

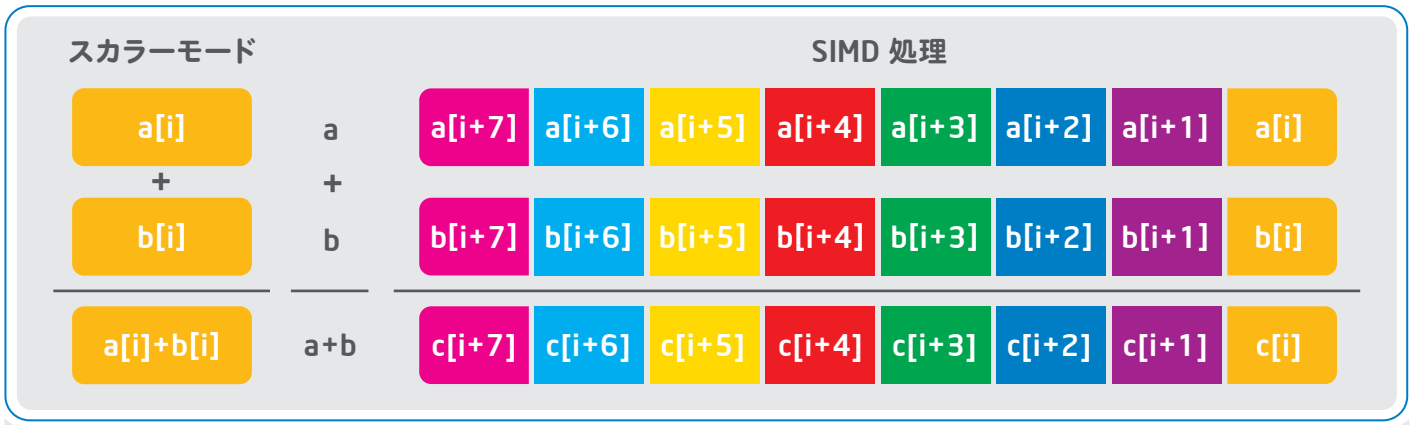


図 1: 配列要素に適用された SIMD 加算

「ベクトル化により、単一プロセッサー・コア内で並列にデータを処理することが可能になり、パフォーマンスが大幅に向上します。ベクトル化は、スレッド化やクラスター並列化と組み合わせることもできます... 推奨されるいずれかの手法を使用してコードをベクトル化することで、現在のパフォーマンスが改善し、さらに将来のスケーリングが可能になる (フォワード・スケーリング) という 2 つのメリットが得られます。」

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

図 1 に示すように、ベクトル化は通常、ループで処理される配列やベクトルデータ要素に対して行われます。加算以外にも、多くの算術関数、論理演算、文字列操作用の SIMD 命令が用意されています。SIMD 命令セットは、大規模なデータセットを処理するアプリケーション向けに設計されています。科学、工学、金融、メディア、グラフィック分野はもちろん、これまでの説明に当てはまるアプリケーションはすべてベクトル化の候補です。

コードをベクトル化する方法

開発者はアプリケーションにおいてさまざまな方法で SIMD 命令を利用することができます。従来のベクトル化は、アセンブリー・コードやコンパイラーの組込み関数を使用して、SIMD 命令とレジスターを直接操作して行われていました。この手法では、開発者が SIMD アーキテクチャーを理解し、プロセッサーに合わせてコードを手動でチューニングする必要があります。さらに、コードの開発と維持に多大な労力がかかるだけでなく、アセンブリーや組込み関数でベクトル化を行うと、コードを別のコンパイラーに単純に移植することはできません。対象とする SIMD ハードウェアごとにコードを記述する必要があるため、この手法によるベクトル化はスケールフォワードではなく、新しいプロセッサーが出るたびに再度実装が必要になります。

幸い、今日ではより少ない労力でスケールフォワード可能なベクトル化の手法を選択することができます。下記のテクニックはベクトル化を行うコンパイラーに依存します。ここでは、C++ および Fortran で利用可能なインテル® コンパイラーを使用することを推奨します。インテル® コンパイラーには高度なベクトライザーが含まれており、ベクトル化をサポートするさまざまなオプション、レポート、拡張機能が用意されています。「6 ステップのベクトル化」プロセスでは、これらの機能のいくつかを利用します。

スケールフォワードの手法

下記のいずれかの手法でコードをベクトル化すると、将来のアーキテクチャーやプロセッサーでも変更を加えることなくベクトル化が行われます。将来のアーキテクチャーでコンパイルを行うと、インテル® コンパイラー（またはこの手法を完全にサポートするコンパイラー）は、適切なベクトル化の方法を選択します。

- インテル® コンパイラーの自動ベクトライザーを使用する：有効な場合、コンパイラーの自動ベクトライザーは、ソースコードを変更することなくループをベクトル化できないかを調べます。ベクトル化フレンドリーなコードでは、ほかの手法は必要ありません。ただし、ループを並列に操作しても安全であることが証明できなければ、コンパイラーはループをベクトル化しません。そのため、この手法に下記のいずれかのテクニックを組み合わせてベクトル化を行うことになります。
- インテル® Cilk™ Plus で提供される高レベルの構造を使用する：インテル® Cilk™ Plus は、並列化とベクトル化をサポートする C/C++ 用の言語拡張です（一部は Fortran にも対応）。現在、インテル® コンパイラーでは完全に、GCC* では部分的にサポートされています。インテル® Cilk™ Plus はオープン・スタンダードです（詳細は、cilk.com を参照）。インテル® Cilk™ Plus は、コードに追加してベクトル化が必要なことをコンパイラーに知らせる、さまざまな構造を提供します。これらの構造の多くは、表記を変更したりプラグマを追加するだけで、コードに簡単に実装することができます。
- 高レベルの Fortran 構造を使用する：Fortran には、配列表記（アレイ・ノーテーション）や DO CONCURRENT ループのようなベクトル化フレンドリーな機能が含まれています。インテル® Cilk™ Plus のベクトル化指示句（!DIR$ SIMD）は、Fortran でも利用可能です。インテル® Cilk™ Plus と同様に、これらの構造はベクトル化できる部分をコンパイラーに知らせるために使用されます。

BLOG highlights

インテル® Cilk™ Plus プログラムのシリアル同値性

インテル コーポレーション
ROBERT GEVA

C++ コミュニティーには、機能をライブラリーとして追加し、言語キーワードの追加を避ける傾向があります。これらの傾向と一致しているのは、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）と Microsoft* Parallel Patterns Library (PPL) です。さて、ここで 1 つの疑問が生じます。インテル® Cilk™ Plus はなぜライブラリーではなく言語拡張として実装されたのでしょうか。

1 つの答えは、言語がコンパイラーによって実装されることで、コンパイラーは一定の保証ができるようになるためです。その保証の 1 つにシリアル等価性があります。並列化の 3 つのキーワードを使用するインテル® Cilk™ Plus プログラムにはすべて、定義済みのシリアルエリジョン（省略コード）があります。シリアルエリジョンは、cilk_spawn と cilk_sync をぞれぞれ空白に、cilk_for を for キーワードに置換して定義されます。明らかに、インテル® Cilk™ Plus プログラムのシリアルエリジョンは有効な C/C++ プログラムです。

論理的に並列な 2 つのストランドが同じメモリー位置にアクセスして、少なくともその 1 つがそのメモリー位置を修正する場合、プログラムには決定性競合があります。インテル® Cilk™ Plus の並列プログラムに決定性競合がなければ、シリアルエリジョンと同じ結果になるでしょう。コンパイラーがシリアル同値性を保証することで何がもたらされるのでしょうか。次のようなコードについて考えてみます...

この記事の続きはこちら（英語）でご覧になれます。

Go-Parallel.com（英語）でご覧になれます。
Go Parallel では、この他にも次のような関連トピックのブログをご覧いただけます：Translating Multicore Power into Application Performance（マルチコアのパワーでアプリケーション・パフォーマンスを引き出そう）。

アプリケーションをベクトル化する 6 ステップのプロセス

前述の手法とインテル® コンパイラーは、アプリケーションをベクトル化する 6 ステップのプロセスの一部です。アプリケーション全体、あるいはその一部に、この手法を試すことができます。このプロセスの実行を支援するため、各ステップを詳細に説明したベクトル化ツールキット (オンライン) を用意しました。ツールキットには、関連情報へのリンクも含まれています。ツールキットについては、http://software.intel.com/en-us/articles/vectorization-toolkit/ (英語) を参照してください。

ステップ 1: 基準となるリリースビルドのパフォーマンスを測定する

まず、ベクトル化による効果があったかどうか分かるように、パフォーマンスの基準が必要です。さらに、目標とするパフォーマンスのレベルを設定することで、目標に到達したかどうかを知ることができます。

パフォーマンスの測定には、デバッグビルドではなくリリースビルドを使用すべきです。リリースビルドには最終的なアプリケーションで使用するすべての最適化が含まれていて、hotspot や実行されるコードがデバッグビルドと異なる場合があるためです。例えば、リリースビルドでは、ベクトル化の候補となる hotspot 内のループを最適化してしまうことがあります。

インテル® コンパイラーでは、リリースビルドがデフォルト設定です。最適化を無効にするには、Windows* の場合はデバッグビルド (または -Zi オプション) 、Linux*/Mac OS* X の場合は -Od オプションを使って、明示的に指定します。インテル® コンパイラーを使用している場合は、自動ベクトライザーが有効になるように、最適化レベル 2 または 3 (-O2 または -O3) を指定していることを確認してください。

ステップ 2: インテル® VTune™ Amplifier XE を使用して hotspot を特定する

パフォーマンス・プロファイラーであるインテル® VTune™ Amplifier XE を使用して、アプリケーションで最も時間を費やしている関数を特定します。ライト hotspot 解析タイプ (システム全体をプロファイル) でも必要な結果が得られますが、hotspot 解析タイプ (アプリケーションのみをプロファイル) を推奨します。

最も時間を費やしているコード領域を特定することで、最も効果的にパフォーマンスを向上できる領域に最適化の労力を集中できます。一般に、アプリケーションの実行時間の 10% 以上を占めている hotspot を解消すれば十分です。次のステップで対象にする hotspot をメモしておきます。

ステップ 3: インテル® コンパイラーのベクトル化レポートを使用してループの候補を決定する

インテル® コンパイラーのベクトル化レポートは、コードの各ループがベクトル化されたかどうかを示します。自動ベクトライザーが有効になるように、最適化レベル 2 または 3 (-O2 または -O3) を指定していることを確認します。ベクトル化レポートを実行して、ステップ 2 で特定した hotspot に対するレポートを確認します。hotspot 内にベクトル化していないループがある場合、並列のデータ (つまり配列) で数学演算、データ処理、文字列操作を行っていないか確認します。これらの処理を行っている場合は、ベクトル化の候補となります。ベクトル化の候補が見つかったら、ステップ 4 に進みます。

ベクトル化レポートを実行するには、-vec-report2 (Linux* および Mac OS* X) または /Qvec-report2 (Windows*) オプションを指定します。

インテル® コンパイラーはコードの一部だけで使用することもでき、ネイティブ・コンパイラーである gcc (Linux* および Mac OS* X) および Microsoft* Visual Studio* (Windows*) と互換性があります。

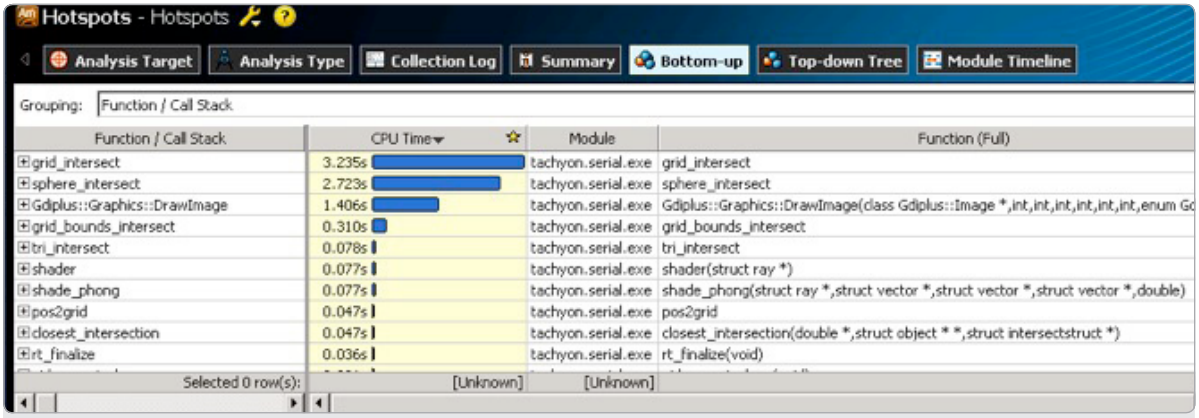


図 2:インテル® VTune™ Amplifier XE の hotspot ビュー

```
.\main.cpp(30): 警告 :ループがベクトル化されました。
.\scalar_dep.cpp(80): 警告 :ループがベクトル化されました。
.\main.cpp(47): 警告 : ループはベクトル化されませんでした: 内部ループではありません。
.): 警告 : ループはベクトル化されませんでした: 非標準のループはベクトル化候補ではありません。
.): 警告 : ループはベクトル化されませんでした: 非標準のループはベクトル化候補ではありません。
.): 警告 : ループはベクトル化されませんでした: ベクトル依存関係が存在しています。
.): 警告 : ループはベクトル化されませんでした: 内部ループではありません。
.): 警告 : ループはベクトル化されませんでした: ベクトル依存関係が存在しています。
.): 警告 : ループはベクトル化されませんでした: 内部ループではありません。
.): 警告 : ループはベクトル化されませんでした: ベクトル依存関係が存在しています。
.): 警告 : ループはベクトル化されませんでした: 内部ループではありません。
```

図 3: インテル® コンパイラーのベクトル化レポートの出力

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

ステップ 4: インテル® コンパイラーの GAP レポートとツールキットのリソースを使用してアドバイスを得る

インテル® コンパイラーのガイド付き自動並列化 (GAP) レポートを実行して、ステップ 3 で候補にしたループをベクトル化する方法についてコンパイラーのアドバイスを確認します。アドバイスを検討し、必要に応じてツールキットのリソースを参照します。

-guide (Linux* および Mac OS* X) または /Qguide (Windows*) オプションを指定して、GAP レポートを実行します。

注 : 必要な場合は、アプリケーションの一部でインテル® コンパイラーを使用することができます。

```
for (i=0; i<n; i++) {

  if (A[i] > 0) { b=A[i]; A[i] = 1 / A[i]; }
  if (A[i] > 1) { A[i] += b;}
}
```

図 4: 非ベクトル化ループの例

```
scalar_dep.cpp(80): 警告 #30515:(VECT) 変数 “b” に、ループ本体の最初 (行 80) で値を代入します。これにより、ループがベクトル化されます。[確認] プログラムにおいてループの各反復で変数 “b” の読み込みを行う場合は、変数がその反復以前に定義されていなければなりません。
```

図 5: 図 4 のループの GAP アドバイス

ステップ 5: GAP のアドバイスとその他の推奨事項 (要素関数や配列表記の使用など) を実装する

ループに対する GAP レポートのアドバイスのうち、可能なものを実装します。

レポートは、コードの変更をアドバイスすることがあります。コードを変更する場合は、コードを変更しても「安全」であること、つまり、ループのセマンティクスや安全性に影響がないことを確認します。依存性がないことを保証する 1 つの方法は、ループを逆に実行したときに結果が変わるかどうかをチェックすることです。ランダムな順序でループを実行したときの結果を確認するのも 1 つの方法です。結果が変わる場合、ループには依存性があり、ベクトル化は「安全」ではありません。ただし、この場合でも、依存性を排除してからベクトル化を行うことは可能です。

ソースを変更して、適切なベクトル化が行われるように、コンパイラーに情報を知らせるか、ループを最適化します。

このとき、インテル® Cilk™ Plus または Fortran により提供される高レベルの構造を使用することができます。利用可能な構造の詳細は、オンラインのベクトル化ツールキットをご覧ください。

ステップ 6: 繰り返す !

パフォーマンスが目標に到達するか、hotspot に候補が見つからなくなるまで、プロセスを繰り返します。

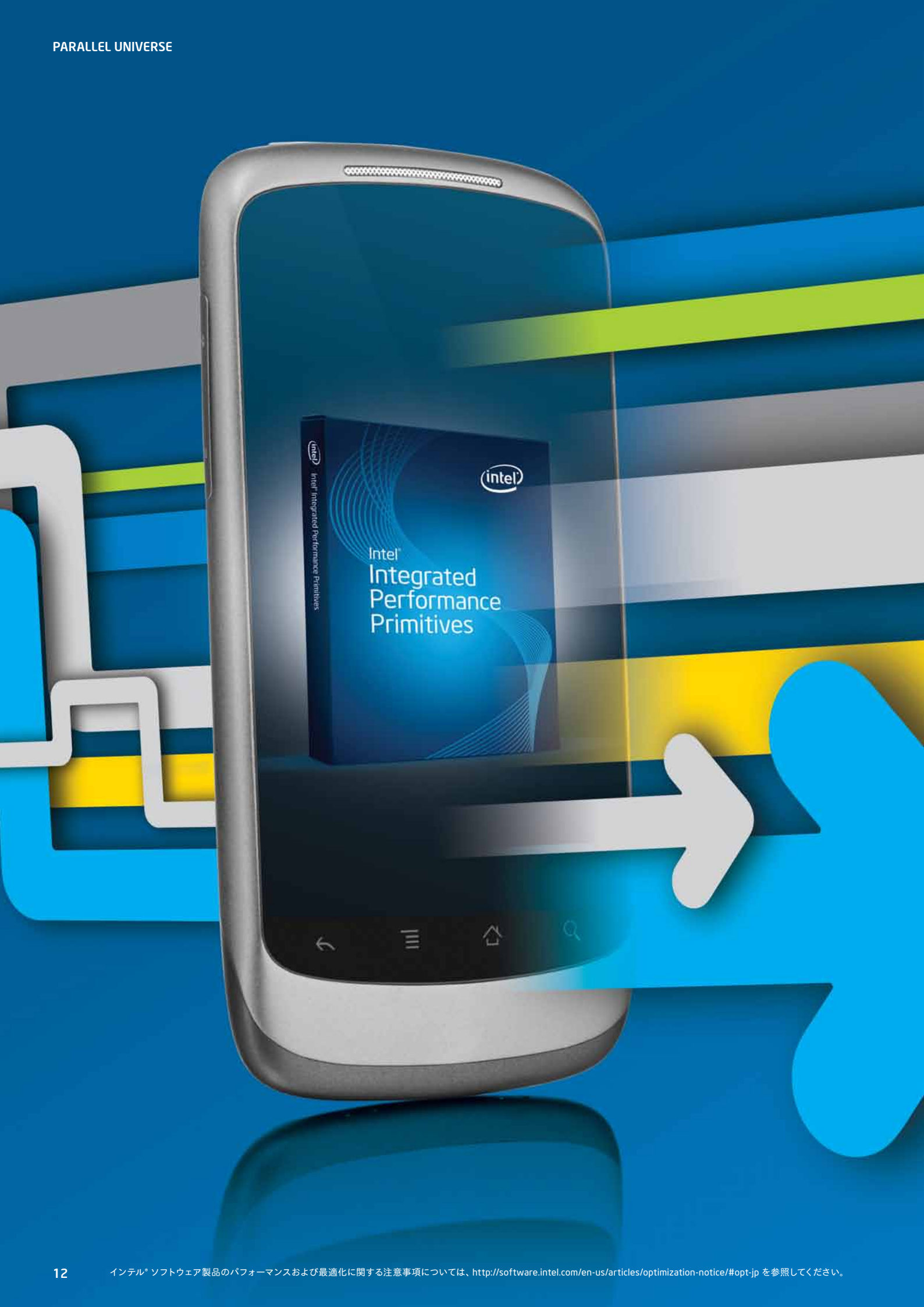
```
for (i=0; i<n; i++) {
      b = A[i];
  if (A[i] > 0) {A[i] = 1 / A[i];}
  if (A[i] > 1) {A[i] += b;}
}
```

図 6: 図 4 のループをベクトル化したもの

まとめ

ベクトル化により、単一プロセッサー・コア内で並列にデータを処理することが可能になり、パフォーマンスが大幅に向上します。ベクトル化は、スレッド化やクラスター並列化と組み合わせることもできます。インテル® Xeon® プロセッサー・ファミリーやインテル® Core™ プロセッサー・ファミリーのような最新のインテル® プロセッサーのパフォーマンスを向上するには、ベクトル化が不可欠です。将来は、これらのプロセッサーに加えて、インテル® メニー・インテグレーテッド・コア (インテル® MIC) アーキテクチャーでもパフォーマンス向上の鍵を握ることになるでしょう。

推奨されるいずれかの手法を使用してコードをベクトル化することで、現在のパフォーマンスが改善し、さらに将来のスケーリングが可能になる (フォワード・スケーリング) という 2 つのメリットが得られます。ベクトル化のための最新のアドバイス、プロセス、リソースについては、ベクトル化ツールキット (英語) サイトを参照してください。また、「将来も通用するアプリケーションの生成」(英語) Web セミナーも是非ご覧ください。



インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

さまざまな環境に応じた 並列計算

インテル® Atom™ プロセッサーでインテル® 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）を活用する

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
Robert Mueller、Noah Clemons、Paul Fischer

インテル® IPP のようなパフォーマンス・ライブラリーが、スモール・フォーム・ファクター・デバイス向けのアプリケーションにおいて、データ集約型タスクの計算実行フローの合理化と統一にどのように役立っているかを紹介します。

インテル® Atom™ プロセッサー・ベースのスモール・フォーム・ファクター・デバイス (インテリジェント・システム) で動作するアプリケーションにパフォーマンス・ライブラリー関数を利用することは、データ集約型タスクの計算実行フローを合理化して統一する優れた方法です。その結果、データストリームのタイミング問題のリスクや、いわゆるハイゼンバグと呼ばれる現象を最小限に抑えられます。ハイゼンバグは、システム・オン・チップ (SoC) コンポーネントやデジタル・シグナル・プロセッサー (DSP) デバイスの微妙なタイミングのずれによって発生する、再現の難しい (多くの場合はランダムな) ランタイム問題です。

インテル® IPP のようなパフォーマンス・ライブラリーには、信号処理、画像処理、ビデオ/オーディオのエンコーディング/デコーディング、暗号化、データ圧縮、音声コーディング、コンピューター・ビジョンを含む、一般的な関数向けに高度に最適化されたアルゴリズムとコードが含まれます。高度な命令セットは、特定のアプリケーション向けに特別に用意された新しいプロセッサーの機能を開発者が活用できるように支援します。開発者は、あたかも低消費電力デバイスまたは組込みデバイス向けの計算を行うブラックボックスのようにインテル® IPP を呼び出し、データを渡して結果を受け取ります。このようにインテル® IPP を使用することで、特定の計算タスク用に作成された多くの処理を代わりに行うことが可能になります。インテル® IPP は、インテリジェント・システムでインテル® Atom™ プロセッサーが利用されるさまざまな領域 (図 1) に対応しています。

高度に最適化されたパフォーマンス・ライブラリーなしで十分な性能を得るには、大量の計算が行われる関数を開発者が注意深く手動で最適化する必要があります。この最適化プロセスは、複雑で時間がかかるだけでなく、プロセッサーの世代が新しくなるたびに更新しなければなりません。インテリジェント・システムは長期間利用されることが多く、関数を手動で最適化することは保守の負担も大きくします。

図 1 に示すように、信号処理と高度なベクトル演算は、異なる種類のインテリジェント・システムで広く利用されている 2 つの関数ドメインです。DSP は、これらの種類の計算タスクを行う汎用プロセッサーを支援するのによく利用されます。DSP は、独自の定義済みアプリケーション・インターフェイスとライブラリー関数セットとともに提供されます。通常は、汎用目的のタスクには適していません。DSP は、基本的な数学演算 (加算、減算、乗算、除算) を高速に実行できるように設計されています。DSP のレパートリーには、畳み込み、内積、その他の複数オペランドの数学演算で頻繁に使用される行列評価に特化した非常に高速な積和演算 (MAC) 命令のセットが含まれます。DSP アプリケーションのコードの多くを占める MAC 命令は、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) に相当します。DSP の MAC 命令のように、これらのインテル® SSSE3 命令は、ベクトルと配列データで非常に効率良く数値演算を実行します。DSP とは異なり、インテル® Atom™ プロセッサーの SIMD 命令は、すべての計算が同じプロセッサーで実行され、1 つにまとめられた論理実行ストリームの一部であるため、複素ベクトルと配列数学アルゴリズムを使用してアプリケーションに簡単に統合できます。

例えば、画像の各ピクセルに定数値を加えて (または引いて) 画像の明るさを変更するアルゴリズムは、メモリーから RGB 値を読み取り、オフセットを加えて (または引いて)、新しいピクセル値をメモリーに書き込む必要があります。DSP コプロセッサーを使用する場合、画像データを DSP 向けにパックし (DSP によってアクセス可能なメモリー領域に配置し)、変換アルゴリズムを実行するようにシグナルを送信した後、汎用プロセッサーに処理を戻す必要があります。SIMD 命令に対応した汎用プロセッサーを利用すると、データセットをパックし、シグナルを送信して、返すという一連のプロセスを単純化できま

	航空宇宙 / 防衛	ネットワーク機器	家電	医療	工業	自動車
データ完全性	■	■		■		
リアリスティック・レンダリング			■	■		
ストリング処理	■	■				
行列 / ベクトル演算	■	■	■	■	■	■
音声コーディング	■	■	■			■
オーディオ・コーディング	■		■			■
ビデオ圧縮	■		■	■		■
画像圧縮	■		■	■		
コンピューター・ビジョン	■				■	
画像カラー変換			■	■		
画像処理			■	■	■	■
信号処理	■	■	■	■	■	■
暗号化	■	■		■		
データ圧縮	■	■		■		

図 1: 組込み分野向けのインテル® IPP の関数ドメイン

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

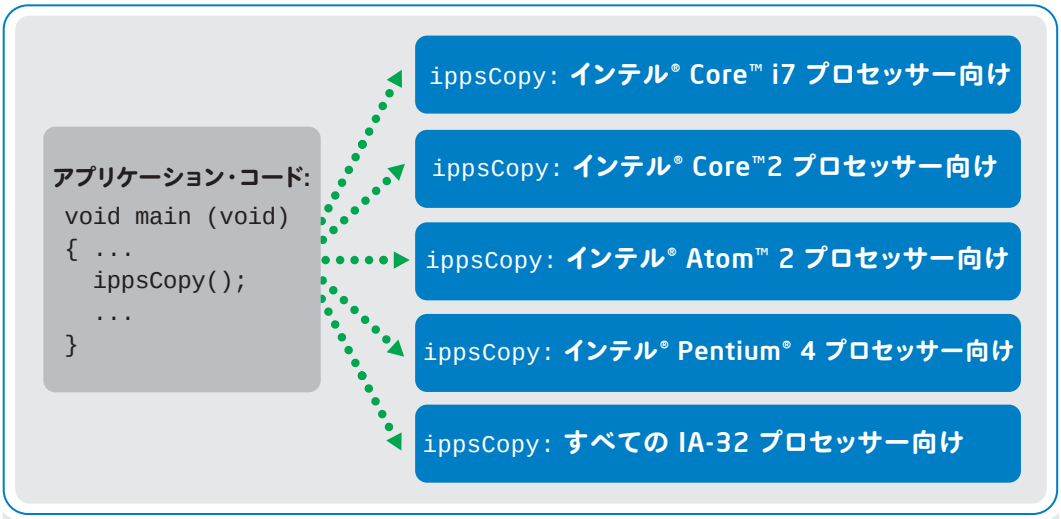


図 2: プロセッサーごとにライブラリーをディスパッチ

す。インテル® IPP ライブラリーのプリミティブは、ぞれぞれの SIMD 命令セット・アーキテクチャーに合わせて最適化されているため、ライブラリーには複数のバージョンのプリミティブが存在します。

インテル® IPP は、さまざまなインテル® アーキテクチャー・ベースのプロセッサーで再利用できます。自動ディスパッチにより、関係のある関数の呼び出しを変更しなくても、開発者が用意したコードは常に対象のアーキテクチャー向けに最適化された実行フローを選択して実行します (図 2)。データ解析と集約にインテル® Core™ プロセッサーを使用し、データの前処理と収集にインテル® Atom™ プロセッサー・ベースの SoC を使用している組込みシステムでは特に有効です。このシナリオでは、同じコードベースがインテル® Atom™ プロセッサーの SoC (フィールド) とインテル® Core™ プロセッサー (中央のデータ集約ポイント) でそれぞれ使用されます。

限られたユーザー・インターフェイスのみを備えたデータ・ストリーミングや I/O 制御用の専用の SoC コンポーネントでは、インテル® Atom™ プロセッサーの最適化や並列性を活用する機会はほとんどないと考えられがちですが、実際には次の処理に活用できます。

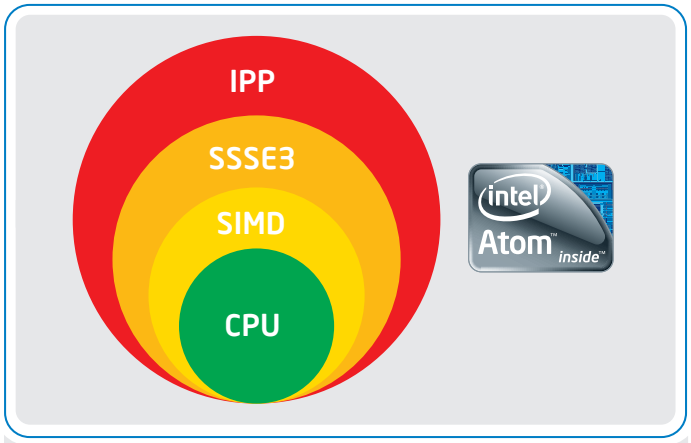


図 3: インテル® IPP は、インテル® Atom™ プロセッサーおよびインテル® SSSE3 セットを活用するようにチューニングされています。

- 異なるアーキテクチャーに基づくヘテロジニアス非同期マルチプロセシング (AMP)。
- 同期マルチプロセシング (SMP)。インテル® ハイパースレッディング・テクノロジーと低消費電力のインテリジェント・システム用に設計された最新世代のインテル® Atom™ プロセッサーで使用されているデュアルコア・デザインを活用します。

この 2 つの概念は、しばしば同じ SoC で共存します。フェイルセーフのリアルタイム要件を備えたコードは、独自のラッパー内で保護され (変更したラウンドロビン・リアルタイム・スケジューラーによって管理される)、残りのオペレーティング・システム (OS) レイヤーとアプリケーション・レイヤーは、標準の SMP マルチプロセシング概念を使用して管理されます。インテル® ハイパースレッディング・テクノロジー対応の 2 つの物理コアを搭載したインテル® Atom™ プロセッサーは、仮想クアッドコア・システムになります。さらに、インテル® Atom™ プロセッサーはインテル® SSSE3 命令セットをサポートします。さまざまなインテル® IPP 関数 (http://software.intel.com/en-us/articles/new-atom-support/) が、インテル® Atom™ プロセッサー・アーキテクチャーとインテル® SSSE3 を活用できるようにチューニングされています (図 3)。

高スループットが求められるアプリケーションは、インテル® SSSE3 ベクトル命令の便利さと、SIMD 処理用の広いベクトルレジスターで複数のデータストリームを並列に実行することにより恩恵が得られます。前述したように、最新のインテル® Atom™ プロセッサーは、最大 4 つの仮想プロセッサー・コアを搭載するように設計されています。スレッド化の際は、この点を考慮する必要があります。すべてのアプリケーションに適した普遍的なスレッド化ソリューションはありませんが、インテル® IPP はスレッドセーフになるように設計されています。

- ライブラリー内のプリミティブは、アプリケーション内の複数のスレッドから同時に呼び出せます。
- 選択するスレッド化モデルによって粒度は異なります。
- インテル® IPP 関数は、OpenMP* により、プロセッサー・コアを直接利用することができます。
- インテル® IPP 関数は、ネイティブスレッド、インテル® Cilk™ Plus、またはその他のインテルの並列モデルを使用する、OS レベルのスレッド化と組み合わせることができます。

インテル® IPP を広範囲に使用するアプリケーションをマルチスレッド化する最も迅速な方法は、ライブラリーにビルトインされている OpenMP* のスレッド化機能を利用することです。大幅にコードを書き直す必要はありません。ただし、スレッド化されているインテル® IPP 関数は 15% から 20% 程度です。ほとんどの場合、最適な結果を達成するには、より高いレベルのスレッド化を行うことが求められます。ライブラリーのプリミティブはスレッドセーフであるため、アプリケーションで直接スレッドを実装し、アプリケーション・スレッド内から直接パフォーマンス・プリミティブを呼び出すことができます。このアプローチではスレッド管理がより良く行えるようになり、アプリケーションの要求に正確に適合するスレッド化が行えます (図 4)。

アプリケーション・レベルでスレッド化する場合は、ライブラリーにビルトインされているスレッド化を無効にすることを推奨します。そうすることで、2 つのスレッド化モデル間のハードウェア・スレッド・リソースの競合が排除され、利用可能なハードウェア・スレッドに対するソフトウェア・スレッドのオーバーサブスクリプションを回避できます。

低消費電力のインテリジェント・システム用のソフトウェアを開発する場合、パフォーマンスと保守性に加えて、フットプリントと消費電力も考慮する必要があります。命令ベースで見た場合、SIMD 命令のほうが非 SIMD 命令よりも消費する電力は多少増加します。しかし、より効率良く実行される (アプリケーションがより短時間で完了する) ため、最終的に作業を完了するまでの合計消費電力は少なくなります。つまり、SIMD 命令を使用することで、バッテリーの消耗を抑えながら、重要なワークロードのパフォーマンスを向上させることができます。

インテリジェント・システムは、制限されたメモリーとストレージで動作しなければならないことがよくあります。そのため、動作中のストレージデバイスやメモリーにおけるアプリケーションのフットプリント (合計バイナリーサイズ) を制御することが非常に重要です。アプリケーションが利用可能なリソース内で実行できない場合、システム全体の信頼性は大きく損なわれてしまいます。

インテル® IPP は、移植性とフットプリント管理のバランスが取れるように、柔軟なリンクモデルを提供します (表 1)。

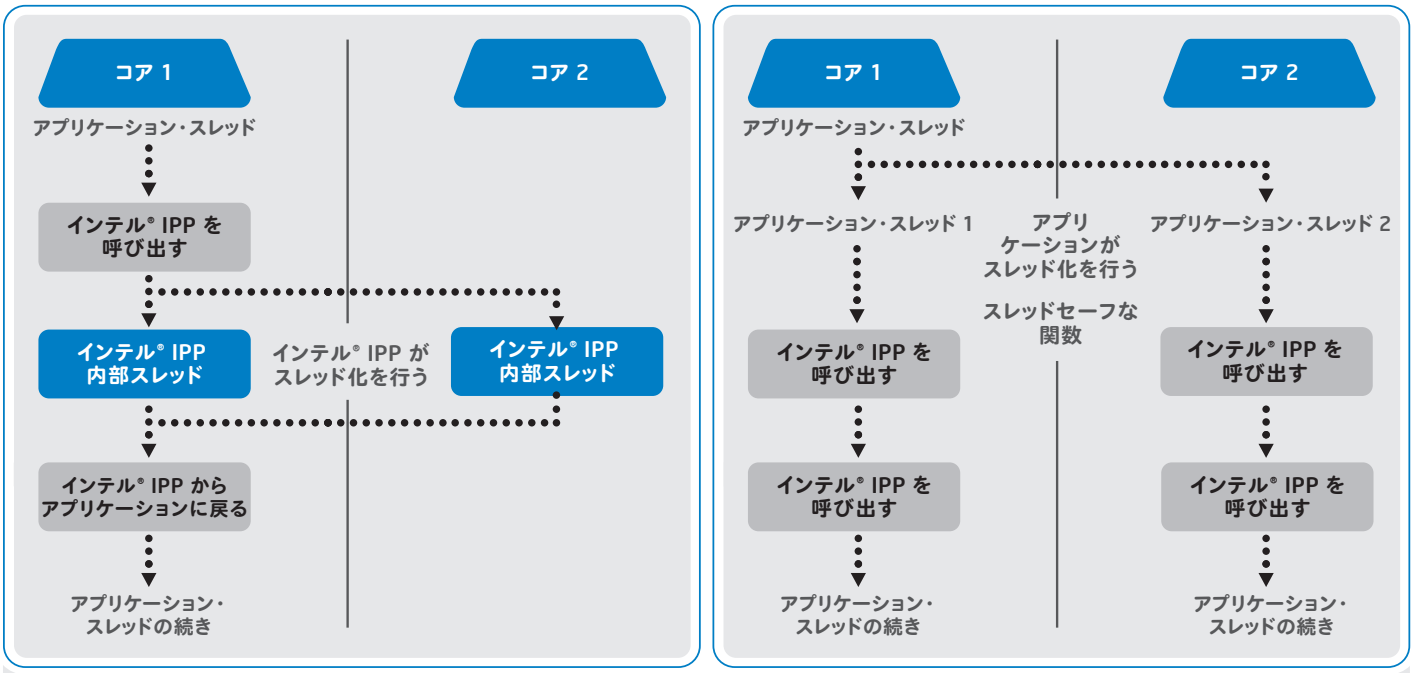


図 4: インテル® IPP を使用した関数レベルのスレッド化とアプリケーション・レベルのスレッド化

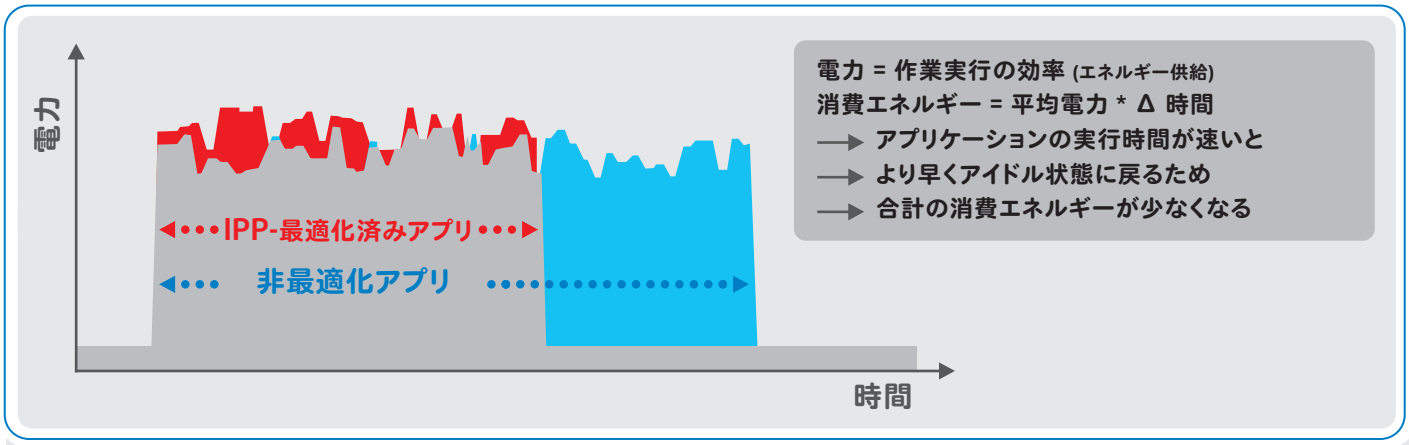


図 5: 最適化されたライブラリーを使用した場合の電力消費の違い

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

スタンダード・ダイナミックとディスパッチ・スタティック・モデルは、インテル® IPP を含むアプリケーションをビルドするときに使用する最も単純なオプションです。スタンダード・ダイナミック・ライブラリーには、すべてのプロセッサーの最適化が含まれており、複数のインテル® IPP ベースのアプリケーションでランタイムコードを共有できるという利点があります。ランタイム・プロセッサーの検出と適切な最適化レイヤーのディスパッチは自動的に行われます。

	スタンダード・ダイナミック	カスタム・ダイナミック	ディスパッチ・スタティック	非ディスパッチ・スタティック
最適化	すべての SIMD セット	すべての SIMD セット	すべての SIMD セット	単一 SIMD セット
配布	実行ファイルとインテル® IPP の標準 DLL	実行ファイルとカスタム DLL	実行ファイルのみ	実行ファイルのみ
ライブラリーの更新	そのまま再配布	リビルドして再配布	アプリケーションを再コンパイルして再配布	カスタム・ライブラリーをリビルドし、アプリケーションを再コンパイルして再配布
実行ファイルのサイズ	小さい	小さい	大きい	中くらい
合計バイナリーサイズ	大きい	中くらい	中くらい	小さい
カーネルモード	未サポート	未サポート	サポート	サポート

表 1: インテル® IPP のリンクモデルの比較

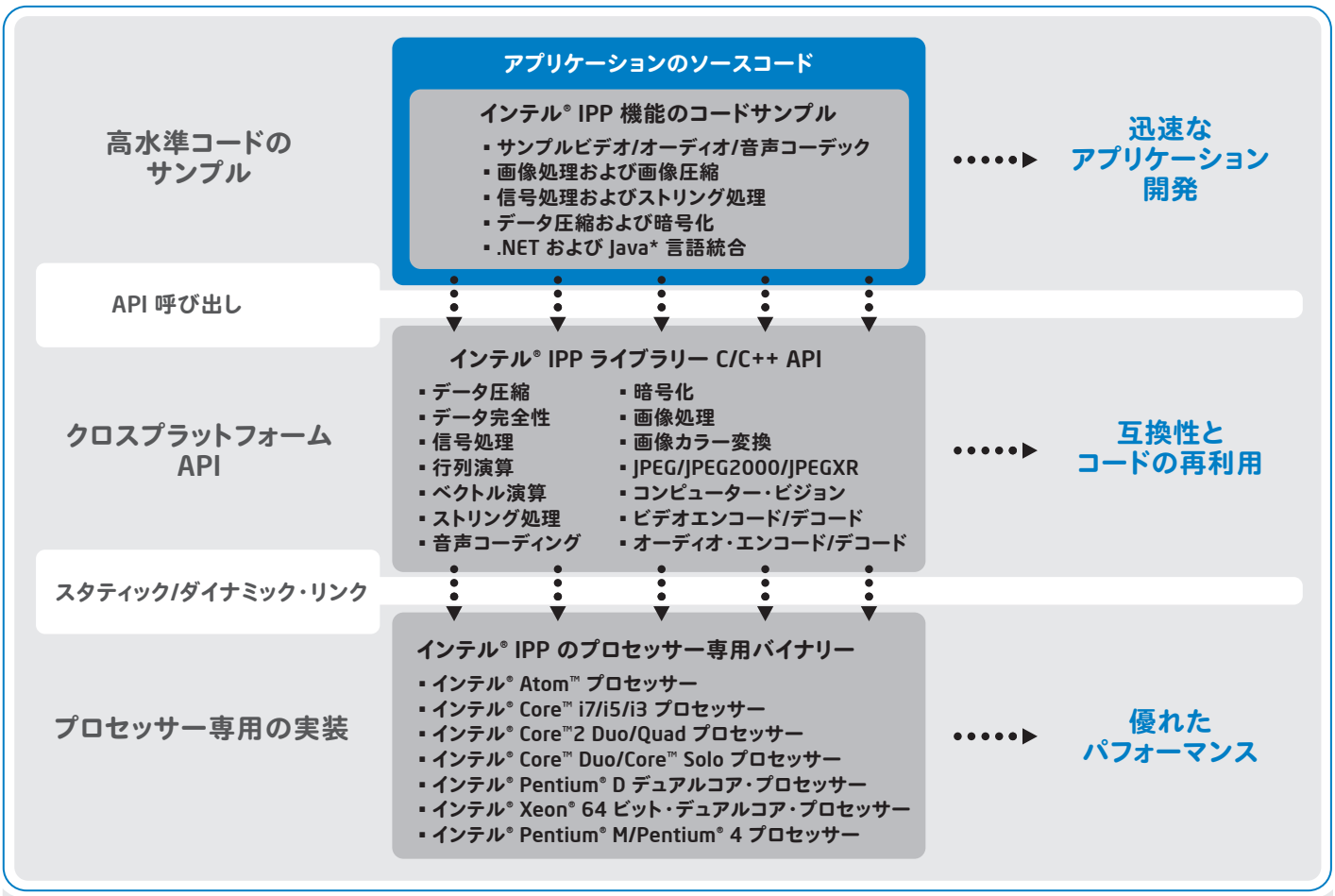


図 6: インテル® IPP の概要。迅速に開発できるように多くのドメインにはサンプルのソースコードが用意されています。

アプリケーションで使用されているインテル® IPP 関数の数が少なく、標準共有ライブラリー・オブジェクトが非常に大きい場合、代わりにカスタム・ダイナミック・ライブラリーを使用します。

合計のバイナリー・フットプリントが最小になるように最適化するには、非ディスパッチ・スタティック・ライブラリーをリンクします。この場合、ターゲット・プロセッサーで必要な最適化レイヤーのみを含む実行ファイルが生成されます。このモデルは、1 つのプロセッサー・タイプと 1 つの SIMD 命令セットに最適化を制限することで、最小のフットプリントを実現します。1 つのプロセッサー・タイプで実行する、ライブラリーを含むアプリケーションを作成する場合は、このリンクモデルが最適です。カーネルモード (リング 0) やデバイスドライバーのアプリケーションにも、このリンクモデルの使用が推奨されます。

組込みインテリジェント・システム用のエコシステムの開発では、デスクトップやサーバーのオペレーティング・システムのように、全体に目を配る必要はありません。SoC は、アプリケーションとユーザー・インターフェイス・レイヤーに Yocto Project* (http://www.yoctoproject.org) や Wind River* Linux* (http://www.windriver.com) をベースとしたカスタム組込み Linux* OS を使用して、チップセットのほかの部分をフェイルセーフなリアルタイム・オペレーティング・システム (FTOS や RTOS) で実行する場合もあります。

現在、インテル® IPP は、Microsoft* Windows*、Linux*、Mac OS* X、QNX Neutrino*、Wind River* VxWorks* の 5 つのオペレーティング・システムをサポートしています。

QNX* および VxWorks* では、インテル® IPP のシングルスレッド・スタティック・バージョンのみ利用できます。インテル® IPP のアプリケーションを「サポートしていない」オペレーティング・システムで実行するには、その OS が前述の 5 つのオペレーティング・システムで定義されるアプリケーション・バイナリー・インターフェイス (ABI) と互換性があり、メモリー割り当てルーチンが標準 C ライブラリーで使用できるか、特別な i_malloc インターフェイスを使用した関連コードでマップできる必要があります。

インテル® IPP 関数実装のアトミックな特長 (ロックやセマフォーがない) は、RTOS の決定的な環境で使用しても安全であることを意味します。RTOS にインテル® IPP ライブラリーを適用した例に、TenAsys 社 (http://www.tenasys.com) の INtime* RTOS for Windows があります。INtime* RTOS は、Windows* ベースのプラットフォームでリアルタイム要件を制御する、Windows* と共存して実行できるように設計された OS です。INtime* RTOS で使用されている ABI は、Windows* ABI と互換性があり、Windows* 互換の関数呼び出し規約を利用しています。これらの RTOS とインテル® IPP を組み合わせて使用することにより、DSP の特性であるディターミニズム (決定性) を兼ね備え、SIMD ベースのデータを高スループットで処理できる環境を構築することができます。

インテル® IPP は、パーソナル・コンピューター分野のネイティブ・アプリケーション開発者だけでなく、アプリケーション・レイヤーとソフトウェア間のインタラクションによってリアルタイム・システムの要件を満たさなければならないインテリジェント・システム開発者のニーズにも対応しています。インテル® IPP をミドルウェア、ドライバー、OS インタラクションの世界にも取り入れることで、リアルタイム要件と主要な実行モデルを備えた組込みデバイスにも利用できます。OS ライブラリーへの依存性がわずかで、柔軟なリンクモデルをサポートしているため、RTOS 特有の環境であっても、Poky-Linux* や MADDE* のような GNU* ベースのクロスビルド環境であっても、組込みクロスビルド環境に簡単に追加できます。

インテリジェント・システムとスモール・フォーム・ファクター・デバイス向けの開発では、ネイティブ開発を行えないことがよくあります。インテル® IPP は、容易にクロスビルド環境に統合し、これらのリアルタイムシステムの多くのフロー要件に適応した、クロスビルドのツールチェーンとともに使用することができます。インテル® IPP を使用すると、組込みインテリジェント・システムでインテル® SSSE3 ベクトル命令とインテル® Atom™ プロセッサーの広いベクトルレジスターを利用できるようになります。開発者は、複雑な SoC アーキテクチャーのアーキテクチャー間データ・ハンドシェイクによるリスクを増加させることなく、ディターミニズムの要件を満たすこともできます。

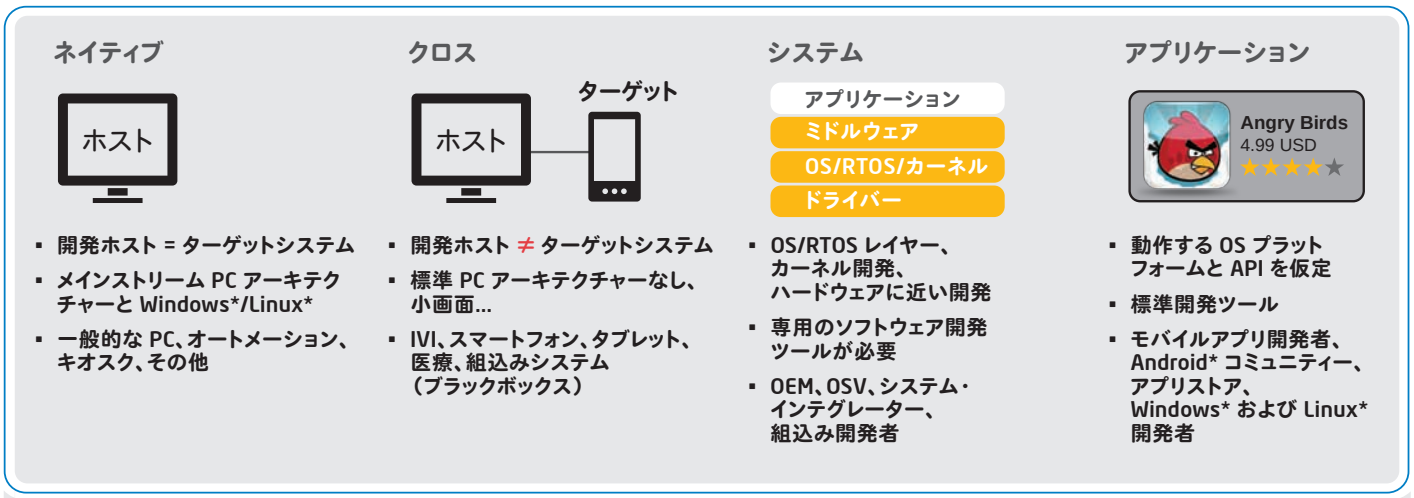


図 7: インテル® IPP は、さまざまな開発状況とソフトウェアで利用することができます。

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

組込みスモール・フォーム・ファクター・デバイスの開発を行うことは、決定的な実行フロー要件を備えたアプリケーションを、システムのソフトウェア・レイヤーおよび OS（または RTOS）スケジューラーとより直接的に接続する必要があることも意味します。ソフトウェア開発ユーティリティーおよびライブラリーは、エンドユーザー・アプリケーションであっても、特別なデータストリームや I/O インターフェイス用のドライバーであっても、あらゆるソフトウェアで動作できるようにしなければなりません。インテル® IPP は、OS への依存性を最小限に抑え、さまざまなモードで動作する定義済みの ABI を備えています（図 7）。アプリケーション・プロセッサー・アーキテクチャーとその長所（ハードウェア・プラットフォームを更新するたびに重要な関数を設計し直す必要がない）を活かしつつ、組込み信号処理およびマルチメディア処理用に高度に最適化された関数をさまざまなソフトウェアに適用できます。

インテル® Atom™ プロセッサーでインテル® IPP を利用する

- インテル® IPP を購入するか、評価版をダウンロードします（http://software.intel.com/en-us/articles/intel-ipp/）。
- インテル® IPP のコードサンプル（http://software.intel.com/en-us/articles/intel-integrated-performance-primitives-code-samples/）を調べて、（実装済み、あるいは実装予定の）アルゴリズムのニーズに合うものがないか確認します。インテル® IPP のコードサンプルは、Windows*、Linux*、Mac OS* オペレーティング・システムで使用できます。各サンプルのビルド方法の説明は、各サンプルのメイン・ディレクトリーにあるサンプル別の ReadMe ファイルに記載されています。一部のサンプルは、doc ディレクトリーにより詳細なドキュメント（通常は PDF ファイル形式）があります。
- ニーズに合うサンプルコードが見つからない場合は、リファレンス・マニュアル（http://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/2011Update/ippxe/ipp_manual_lnx/index.htm）を参照してみてください。マニュアルは、関数ドメイン別、ドメイン内の関数のクラス別に構成されています。インテル® IPP の関数は約 12,000 あり、14 のドメインに分かれています（図 1）。
- 最後に、表 1 を参考にしてリンクオプションを選択します。これらのオプションの詳細は、http://software.intel.com/en-us/articles/introduction-to-linking-with-intel-ipp-70-library/ を参照してください。

BLOG highlights

タスクベースの並列化モデルを使用するパフォーマンス上の利点

インテル コーポレーション
SHANNON CEPEDA

ソフトウェアのパフォーマンス向上の一環として、私はアプリケーションへのスケーラブルな並列化の実装に関するサポートとコンサルティングを行っています。並列化の実装にはさまざまな理由があり、その手法もさまざまです。しかし、このブログではどちらについても触れません。このブログでは、並列化を実装するある特定の手法のパフォーマンスの利点について述べます。幸い、その手法は複数のモデルによってサポートされています。

その手法とは、タスクベースの並列化です。つまり、特定のスレッドやプロセッサー・コアの代わりに、タスク（行われる作業）の点からアルゴリズムを設計することになります。現在タスクを使用する並列化モデルには、インテル® TBB、インテル® Cilk™ Plus、Microsoft* Parallel Patterns Library (PPL)、OpenMP* 3.0 などがあります。これらのタスクベースのライブラリーを使用することで、事前にコーディングされた関数、プラグマ、キーワード、テンプレートを使用して、アプリケーションを並列化できます...

この記事の続きはこちら（英語）でご覧になれます。

Go-Parallel.com（英語）でご覧になれます。
Go Parallel では、この他にも次のような関連トピックのブログをご覧いただけます：Translating Multicore Power into Application Performance（マルチコアのパワーでアプリケーション・パフォーマンスを引き出そう）。

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

データ競合

初期化されていないメモリーへのアクセス

割り当ての不一致

メモリーエラー

ボトルネック

.NET アプリケーションの信頼性とパフォーマンスを高めるツール

.NET 開発者は、アプリケーションのパフォーマンスを向上させて、ハイパフォーマンス・コンピューティングとエンタープライズ・アプリケーションで必要とされるコードの品質と信頼性を高めることができるようになりました。

インテル コーポレーション
ソフトウェア & サービスグループ
Levent Akyil、Asaf Yaffe

開発サイクルの初期にエラーを発見できる開発者フレンドリーなツールを利用する効果は絶大です。リリースした製品に不具合が残されていると、製品の評価を大きく下げるだけでなく、修正に多大な費用がかかります。アプリケーションのスレッドエラー（データ競合およびデッドロック）やメモリーエラー（メモリーリーク）のように発見が困難なエラーを早期に検出できれば、より簡単に修正することができます。信頼性やパフォーマンスをチェックするツールは、問題が含まれるソースコード行、イベントのコールスタックやタイムラインをビジュアルに表示し、ソフトウェアに含まれる問題について分かりやすく示すことで、開発者を支援します。インテル® Parallel Studio XE を利用することで、.NET 開発者は、最も時間を費やしている関数やコード行などの重要なパフォーマンスの問題、スケーラビリティーの問題、同期構造や I/O アクティビティーの待機時間を特定することができます。インテル® Parallel Studio XE は、分岐予測ミス、キャッシュミス、メモリー帯域幅の問題などによって引き起こされる潜在的なマイクロアーキテクチャーのボトルネックを発見します。

この記事では、インテル® Parallel Studio XE に含まれる 2 つのツール（エラー検出ツール インテル® Inspector XE とパフォーマンス分析ツール インテル® VTune™ Amplifier XE）が .NET コード、ネイティブコード、「混合」（.NET およびネイティブ）アプリケーションの開発サイクルでどのように開発者に役立つか、例を挙げて説明します。インテル® Inspector XE とインテル® VTune™ Amplifier XE における現在の .NET のサポートについて言及した後、主な機能を C# アプリケーションで利用する方法を紹介します。

インテル® VTune™ Amplifier XE と インテル® Inspector XE における現在の .NET のサポート

インテル® Inspector XE とインテル® VTune™ Amplifier XE は、純粋な .NET アプリケーションに加えて、マネージドコードとアンマネージド・コードの両方を含む「混合」アプリケーションの解析をサポートしています。

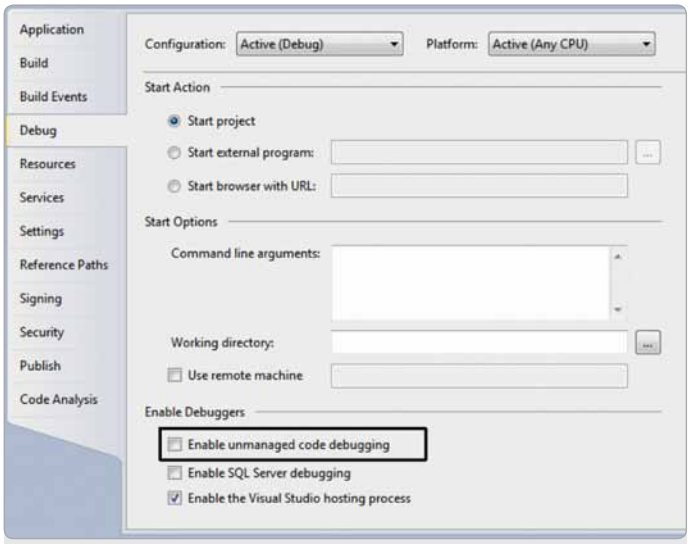


図 1: Microsoft* Visual Studio* の .NET プロジェクト用のデバッグ・プロパティー

インテル® Inspector XE のスレッド・アナライザーは、ネイティブコードと同様の方法で、.NET プログラムの潜在的なデッドロックやデータ競合を検出します。インテル® Inspector XE は、ガベージコレクトしたヒープおよびスタティック・データ領域のオブジェクト割り当てと共有メモリーへのアクセスをモニターして、複数のスレッドによる同一オブジェクト/クラスのデータメンバーへの非同期アクセス（少なくとも 1 つは書き込み操作）を潜在的なデータ競合としてフラグ付けします。インテル® Inspector XE は、.NET 2.0 から .NET 3.5 までのロック API によるデッドロックとロック階層違反を検出することもできます。

インテル® VTune™ Amplifier XE は、近年のプロセッサーで最適なパフォーマンスを発揮できるようにシリアル/並列アプリケーションのきめ細かな調整を支援します。また、.NET 開発者が純粋な .NET アプリケーションや混合アプリケーションのパフォーマンス・ボトルネックを素早く検出できるようにします。インテル® VTune™ Amplifier XE の hotspot 解析は、アプリケーションで最も実行時間を費やしている関数とソースの場所を特定します。コンカレンシー解析およびロックと待機解析は、スレッド間の作業分配とスレッドの同期ポイントをビジュアルに表示するため、並列実行の妨げとなっている作業分配の問題と過度なスレッド同期を開発者は識別することができます。インテル® VTune™ Amplifier XE は、インテル® プロセッサーのパフォーマンス・モニタリング・ユニット（PMU）を使用してプロセッサー・イベントをサンプリングし、アーキテクチャーのボトルネックを特定することで、開発者がマイクロアーキテクチャーのパフォーマンスの問題を把握できるようにサポートします。

インテル® VTune™ Amplifier XE と インテル® Inspector XE で .NET 解析を設定する

ユーザーは、解析する部分として、マネージド部分（「マネージド」モード）、ネイティブ部分（「ネイティブ」モード）、または両方（「混合」モード）を選択できます。これらのモードは以下のようにサポートされています。

- **ネイティブ**モードは、ネイティブコードのデータのみ収集し、マネージドコードのデータは収集しません。
- **マネージド**モードは、マネージドコードのデータのみ収集し、ネイティブコードのデータは収集しません。
- **混合**モードは、必要に応じてネイティブコードとマネージドコード両方のデータを収集します。このオプションは、ネイティブコードの呼び出しを行うマネージド実行ファイルを解析するときに使用します。
- **自動**モードは、ターゲット実行ファイルの種類を自動的に検出します。マネージド・アプリケーションが検出されると混合モードに切り替え、ネイティブ・アプリケーションが検出されるとネイティブモードに切り替えます。

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

解析モードの設定方法は、ツールの使用法によって異なります。コマンドラインからツールを使用する場合は、「-mrte-mode」オプションでモードを指定します。Microsoft* Visual Studio* 内でツールを使用する場合は、アクティブなプロジェクト・タイプに基づいて解析モードが自動的に選択されます。ネイティブ・プロジェクト (C/C++ アプリケーション) の場合、インテル® VTune™ Amplifier XE とインテル® Inspector XE の解析モードはどちらも「ネイティブ」に設定されます。.NET プロジェクト (C# アプリケーション) の場合、インテル® VTune™ Amplifier XE のデフォルトの解析モードは「マネージド」に設定され、インテル® Inspector XE のデフォルトの解析モードは「混合」に設定されます。Visual Studio* の [Debug (デバッグ)] プロパティー・ページを使用して、異なるモードを選択することができます。.NET プロジェクトで「混合」解析モードを有効にするには、[unmanaged code debugging (アンマネージ コード デバッグを有効にする)] オプションをオンにします (図 1)。同様に、ネイティブ・プロジェクトで「混合」解析モードを有効にするには、[Debugger Type (デバッガーの種類)] を [Mixed (混合)] に設定します (図 2)。インテル® VTune™ Amplifier XE またはインテル® Inspector XE のスタンドアロン・グラフィカル・インターフェイスを使用する場合は、[Project Properties (プロジェクトのプロパティー)] ダイアログから解析モードを設定します (図 3 および 4)。

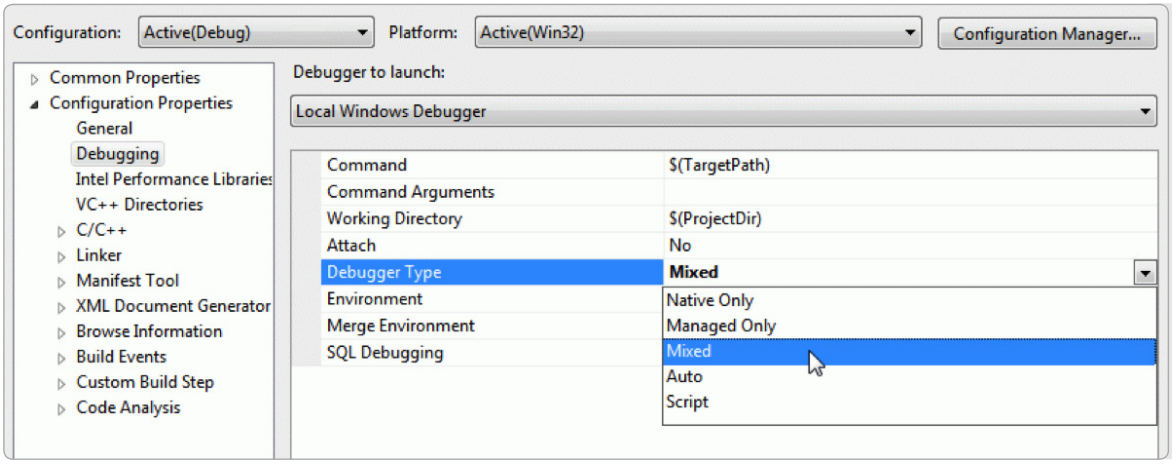


図 2: Microsoft* Visual Studio* のネイティブ・プロジェクト用のデバッグ・プロパティー

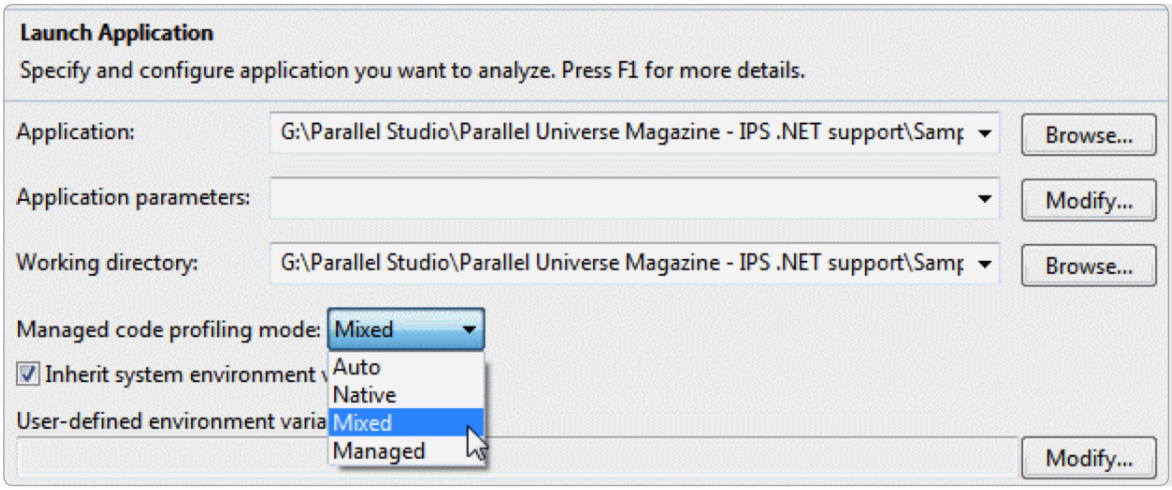


図 3: インテル® VTune™ Amplifier XE のプロジェクト・プロパティー

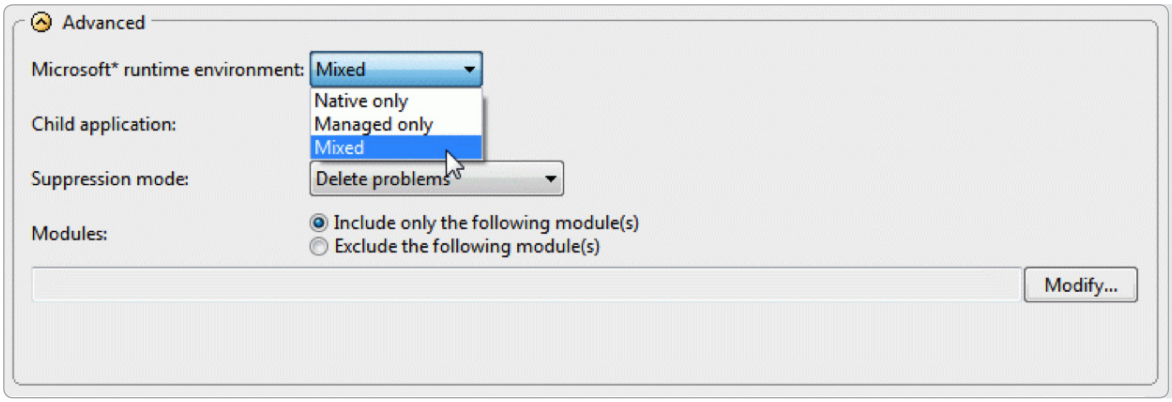


図 4: インテル® Inspector XE のプロジェクト・プロパティー

```
namespace VTuneAmplifierXE.Examples
{
    public class POTENTIAL_MT
    {
        private static Thread[] threads = new Thread[Benchmarks.gThreadCount];
        private static workerThread[] workerThreads = new workerThread[Benchmarks.gThreadCount];
        private static object[] threadparams = new object[Benchmarks.gThreadCount];

        // 信号の開始と完了
        static AutoResetEvent[] goSignals = new AutoResetEvent[Benchmarks.gThreadCount];
        static AutoResetEvent[] doneEvents = new AutoResetEvent[Benchmarks.gThreadCount];

        public static double potential = 0.0;
        public double potentialTotal = 0.0;

        public class workerThread
        {
            private volatile bool stopNow = false;
            private ThreadParameters threadParameters;

            public void RequestStop()
            {
                stopNow = true;
            }

            private void computePot_mt(int tid)
            {
                int start, end;
                double distx, disty, distz, dist;

                start = threadParameters.chunkBegin;
                end = threadParameters.chunkEnd;

                potential = 0.0;

                for (int i = start; i < end; i++)
                {
                    for (int j = 0; j < i - 1; j++)
                    {
                        distx = Math.Pow((r[0][j] - r[0][i]), 2);
                        disty = Math.Pow((r[1][j] - r[1][i]), 2);
                        distz = Math.Pow((r[2][j] - r[2][i]), 2);
                        dist = Math.Sqrt(distx + disty + distz);
                        potential += 1.0 / dist;
                    }
                }
            }
            …

            public void doWork(object parameter)
            {
                threadParameters = (ThreadParameters)parameter;
                Console.WriteLine("Thread:{0} ready to start <{1} - {2}>",
                    threadParameters.tid,
                    threadParameters.chunkBegin,
                    threadParameters.chunkEnd);

                while (!stopNow)
                {
                    threadParameters.goSignal.WaitOne();
                    computePot_mt(threadParameters.tid);
                    threadParameters.eventDone.Set();
                }
                Console.WriteLine("worker thread: terminating gracefully.");
            }
        } // workerThread の最後
        …
    } // POTENTIAL_MT の最後
```

図 5: 質点系の位置エネルギーを計算する C# プログラム

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

サンプルコード

インテル® Inspector XE とインテル® VTune™ Amplifier XE が .NET アプリケーションをどのようにサポートしているかを示すため、ここでは、3 次元の距離に基づいて質点系の位置エネルギーを計算する C# プログラムを使用します。このプログラムは .NET のスレッドプールを使用してスレッド化されており、利用可能なコア数のスレッドを生成します。この記事では、C# スレッドや .NET フレームワークを使用したスレッド化の効率については説明しません。ハイパフォーマンスでスケーラブルな並列アプリケーションを開発する際に、各ツールがスレッド化の問題をどのように特定して開発に役立つかを説明します。

図 5 のコードは、各ワーカースレッドで実行されるアプリケーションの部分を示しています。computePot メソッドで、各スレッドは、スレッドの割り当て識別番号 (tid) でインデックスされた境界を用いて、計算する質点の開始 (start) と終了 (end) を決定します。各スレッドは、反復空間 (start から end) を初期化した後、質点の位置エネルギーの計算を開始します。

インテル® Inspector XE

では、サンプルコードに対してインテル® Inspector XE を実行してみましょう。Visual Studio* の [Tools (ツール)] メニューから、[Intel Inspector XE 2011 (インテル® Inspector XE 2011)] > [New Analysis (新規解析)] を選択します (図 6)。[Configure Analysis Type (解析タイプの設定)] ページで、[Locate Deadlocks and Data Races (デッドロックとデータ競合の特定)] を選択し (図 7)、[Start (開始)] ボタンをクリックしてスレッド正当性解析を開始します。

これを実行すると、データ競合があることが検出されます (図 8)。

[Problems (問題)] ペインに個々の問題がリストされます。[Code Locations (コードの場所)] には、選択した問題と関連しているソースコードの場所が表示されます。[Filters (フィルター)] ペインを使用して、重要度、問題の種類、モジュール、ソースファイルで [Problems (問題)] ビューの表示をフィルタリングすることができます。

問題をダブルクリックすると、[Sources (ソース)] ページ (図 9) に移動して、問題をより詳細に調査することができます。

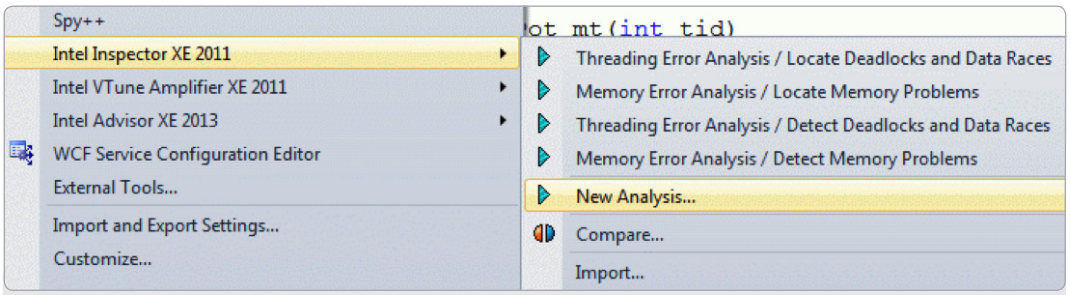


図 6: Visual Studio* から新しいインテル® Inspector XE の解析を開始

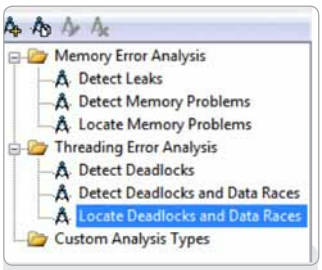


図 7: インテル® Inspector XE の解析タイプの選択

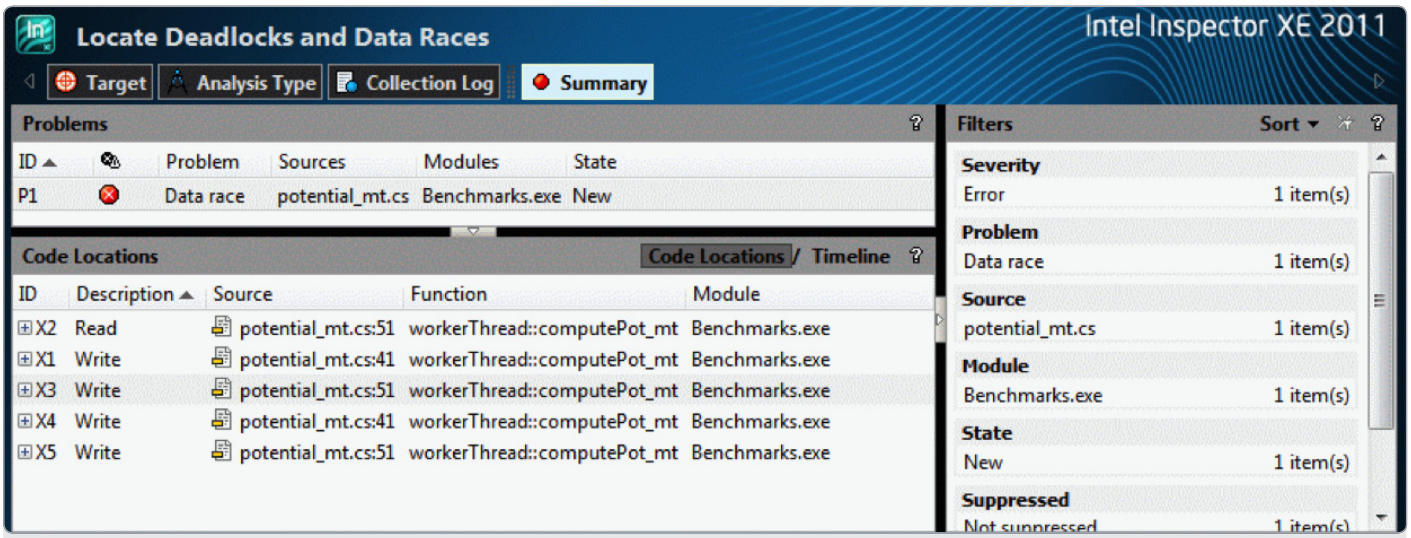


図 8: インテル® Inspector XE 2011 で検出された潜在的なデータ競合

このページでは、共有メモリーへの非同期アクセスを行う典型的な 2 つの実行スレッドとその詳細なコールスタックが表示されます。表示内容を確認することで、スタティック・クラス・メンバー "potential" へ非同期アクセスが行われているかどうか、つまり、典型的なデータ競合が存在するかどうかをすぐに判断することができます。ソース行をダブルクリックして直接ソースコードにジャンプし、問題を修正できます。

このデータ競合の最も単純なソリューションは、"potential" クラスメンバーへのアクセスがすべて、ロックで適切に同期されているかを確認することです。ただし、このソリューションでは並列コードにシリアル領域 (クリティカル・セクション) が追加されるため、パフォーマンスに影響します。より優れたソリューションは、各スレッドのローカル変数に potential のプライベート・コピーを格納し、結果を合計して最終的な値を計算することです。このソリューションでは依存性とスレッド間の同期が少なくなるため、並列コードの速度が向上します。

メモリーチェッカーについて

インテル® Inspector XE のメモリー・アナライザーは、.NET コードに対応しており、マネージドコードとネイティブコードを含む混合アプリケーションのメモリーリークとメモリーエラーの検出に使用することもできます。メモリー解析はアプリケーションのネイティブ部分に対して行われますが (ほとんどの検出可能なエラーは .NET コードとは無関係であるため)、解析結果には、ネイティブコードのメモリーエラーを引き起こした一連の .NET 呼び出しを含む、完全なスタックトレースが表示されます。

インテル® Inspector XE が、.NET コードとネイティブコードが含まれる複雑なアプリケーションの正当性を解析できる、強力で貴重なツールなのは、スレッド化解析とメモリー解析機能の組み合わせによるためです。

インテル® VTune™ Amplifier XE

正当性問題を修正したら、アプリケーションのパフォーマンスの解析を開始しましょう。図 10 は、Visual Studio* 内でコンカレンシー解析を開始する方法を示しています。第 2 世代インテル® Core™ プロセッサー・ファミリーのクアッドコア・プロセッサー (2.5GHz) でアプリケーションを解析した結果サマリーを 図 11 に示します。

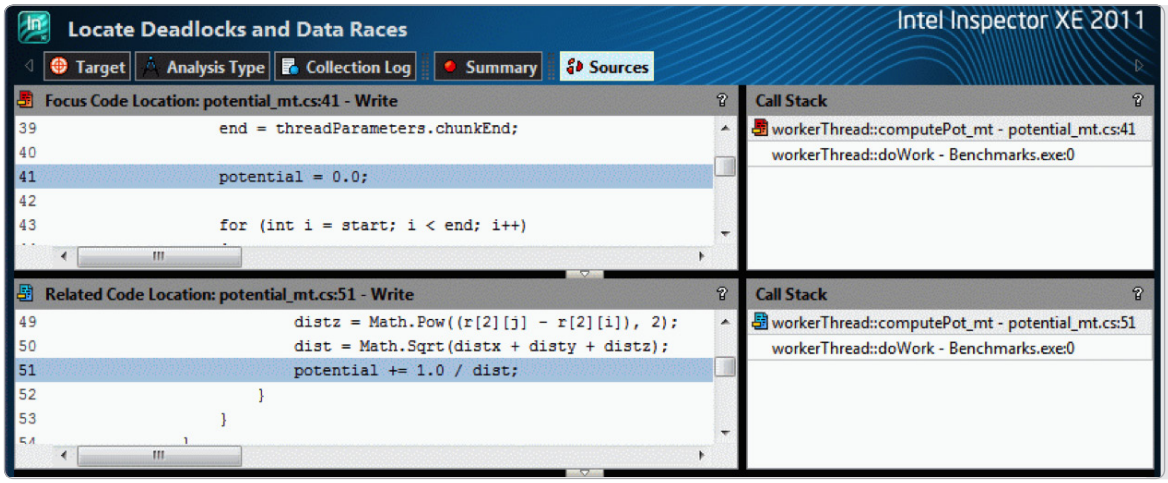


図 9:
データ競合問題の詳細

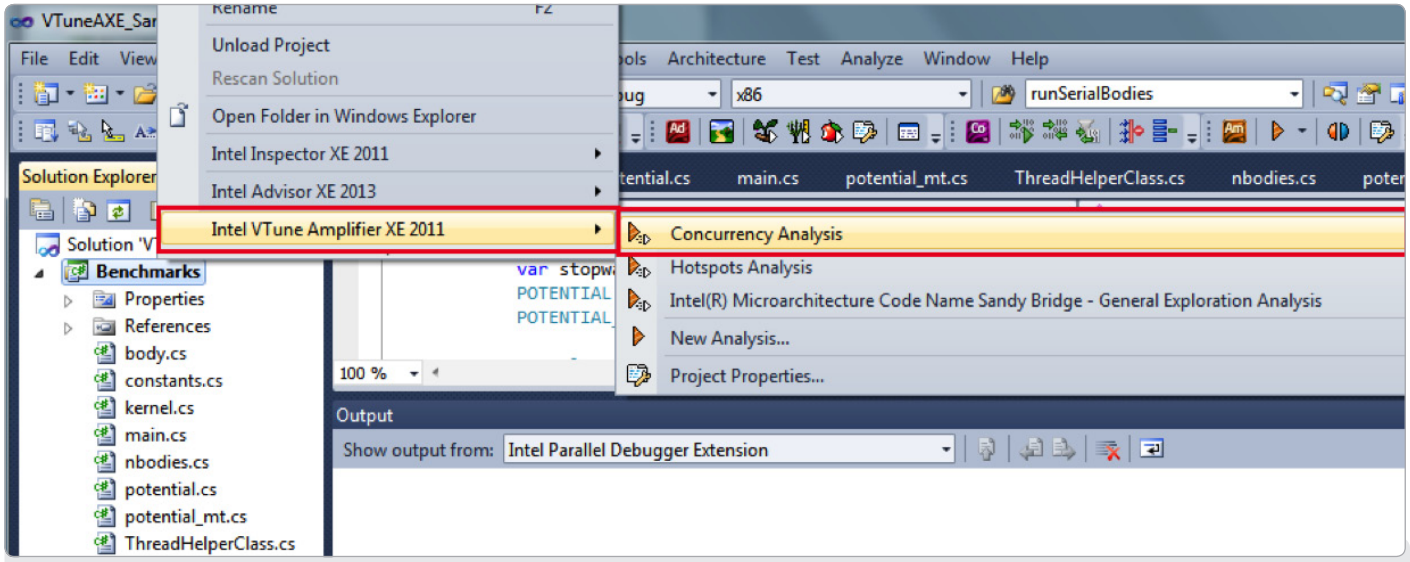
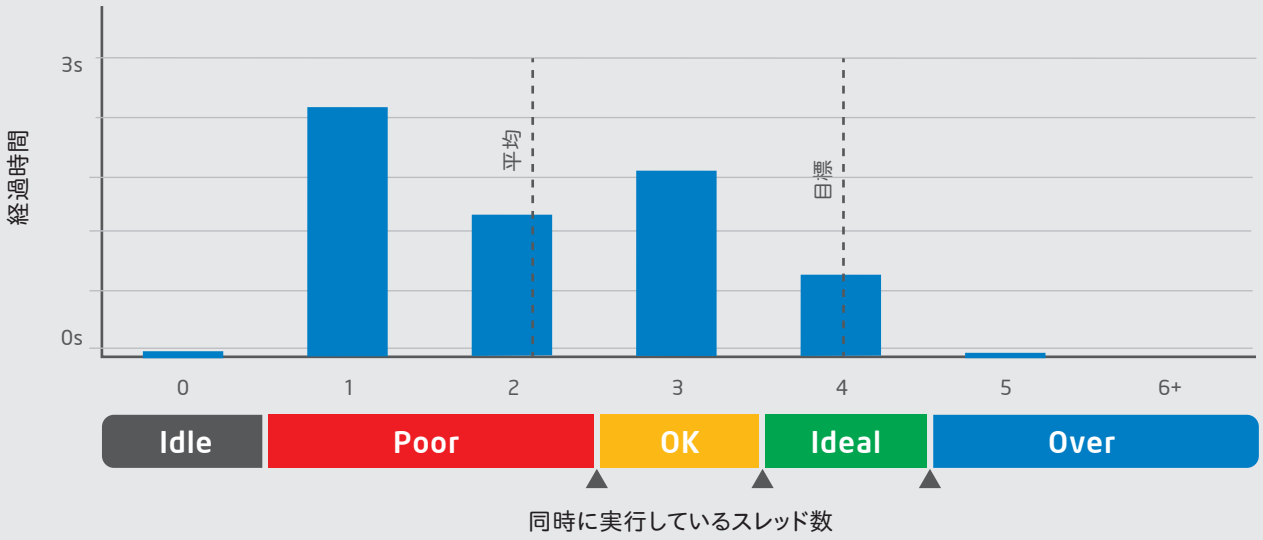


図 10: Visual Studio* 内でインテル® VTune™ Amplifier XE のコンカレンシー解析を開始

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

スレッド・コンカレンシー・ヒストグラム

このヒストグラムは経過時間のブレークダウンを表します。特定の数のスレッドが同時に実行していたウォール時間の割合が表示されます。スレッドが CPU 上で実際に実行していた場合、または OS スケジューラーで実行可能な状態であった場合、スレッドは実行していたと見なされます。スレッド・コンカレンシーは、本質的には待機していないスレッド数を測定するものです。スレッドが実行可能な状態で CPU 時間を消費していない場合、スレッド・コンカレンシーは CPU 使用率よりも高くなります。



CPU 使用率ヒストグラム

このヒストグラムは経過時間のブレークダウンを表します。特定の数の CPU が同時に実行していたウォール時間の割合が表示されます。スレッドが論理 CPU 上でコードを実行している場合、CPU 使用率はスレッド・コンカレンシーよりも高くなります。

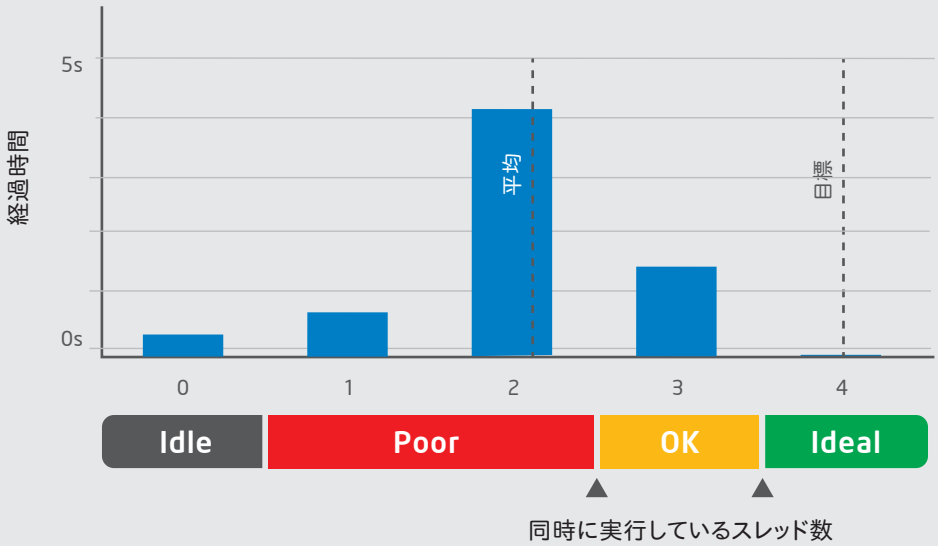


図 11: 解析のスレッド・コンカレンシー・ヒストグラムと CPU 使用率ヒストグラム

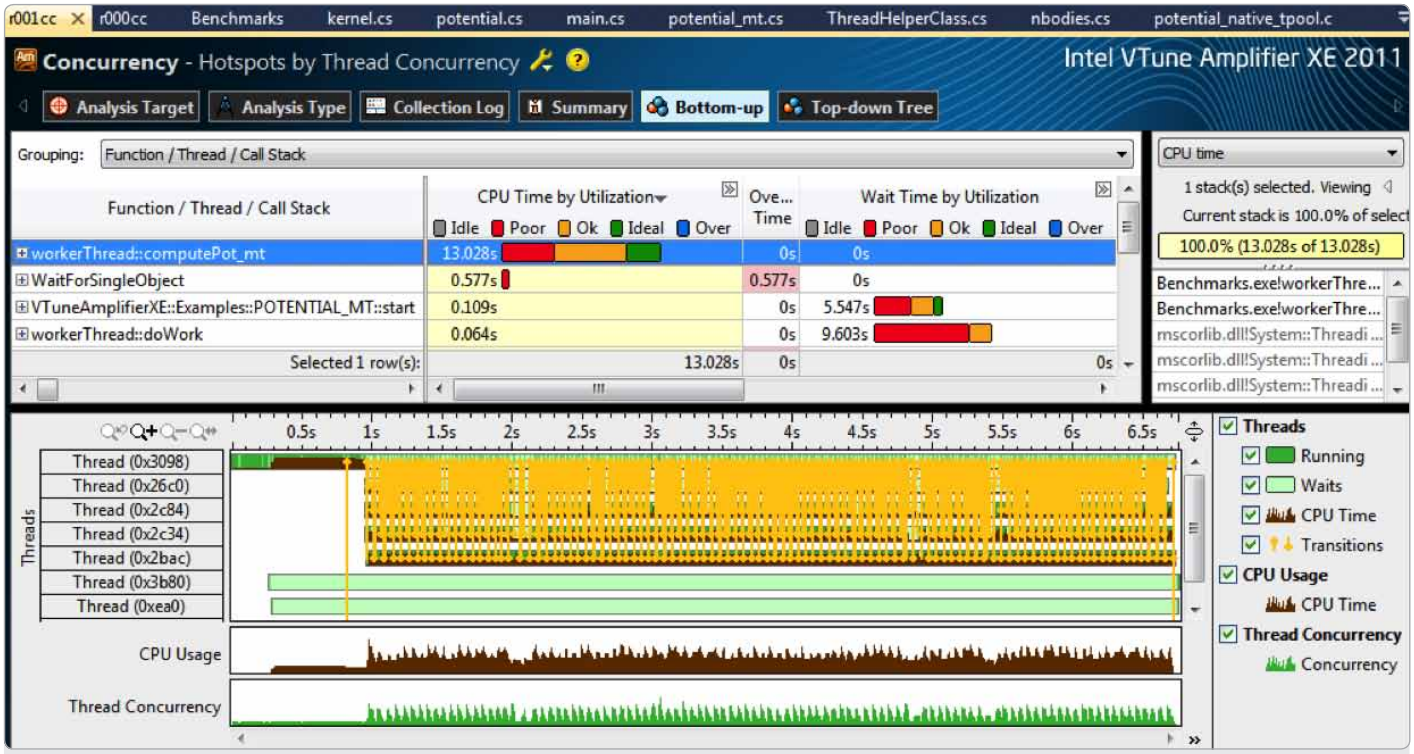


図 12: [Hotspots by Thread Concurrency (スレッド別 hotspot コンカレンシー)] ビュー

図 11 は、アプリケーションが利用可能なすべてのコアを十分に活用していないことを示しています。[Bottom-up (ボトムアップ)] ビュー (図 12) で詳細な結果を確認します。workerThread:: computePot_mt メソッドが CPU 時間のほとんどを費やしていて、その多くの時間の CPU 使用率が Poor (赤) および Ok (オレンジ) になっていることが分かります。

これは、特にこのメソッドが hotspot (つまり、CPU 時間のほとんどを費やしている) であり、スレッド化されているにもかかわらず利用可能なコアをほとんど使用していないことを示します。問題を特定するため、タイムラインを拡大して、このメソッドを実行しているスレッドを調べます。図 13 から、workerThread::computePot_mt メソッドを実行する 4 つのスレッドが費やしている CPU 時間が異なっているために、ロード・インバランスが発生し、コアが効率的に利用されていないことが分かります。ロード・インバランスが発生していると、より多くのコアでアプリケーションを実行したときに適切にスケーリングされないため、修正しておく必要があります。

外側の for ループの実行回数はすべてのスレッドで同じですが、内側のループの実行回数は最後のチャンクを実行するスレッドのほうが多くなり、最初のチャンクを実行するスレッドのほうが少なくなります。スレッドカウントによって反復の循環オフセットを分配することで、ロード・インバランスが修正され、各スレッドは利用可能なコアをより適切に利用するようになります。コンカレンシー解析とその結果は、ロード・インバランスの問題を特定できるだけでなく、アプリケーションのスピードアップにも役立ちます。図 15 のようにコードを変更することで、ロード・インバランスが解消され、すべてのスレッドが効率的に実行されるようになりました (図 16)。

「ロード・インバランスが発生していると、より多くのコアでアプリケーションを実行したときに適切にスケーリングされないため、修正しておく必要があります。」

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

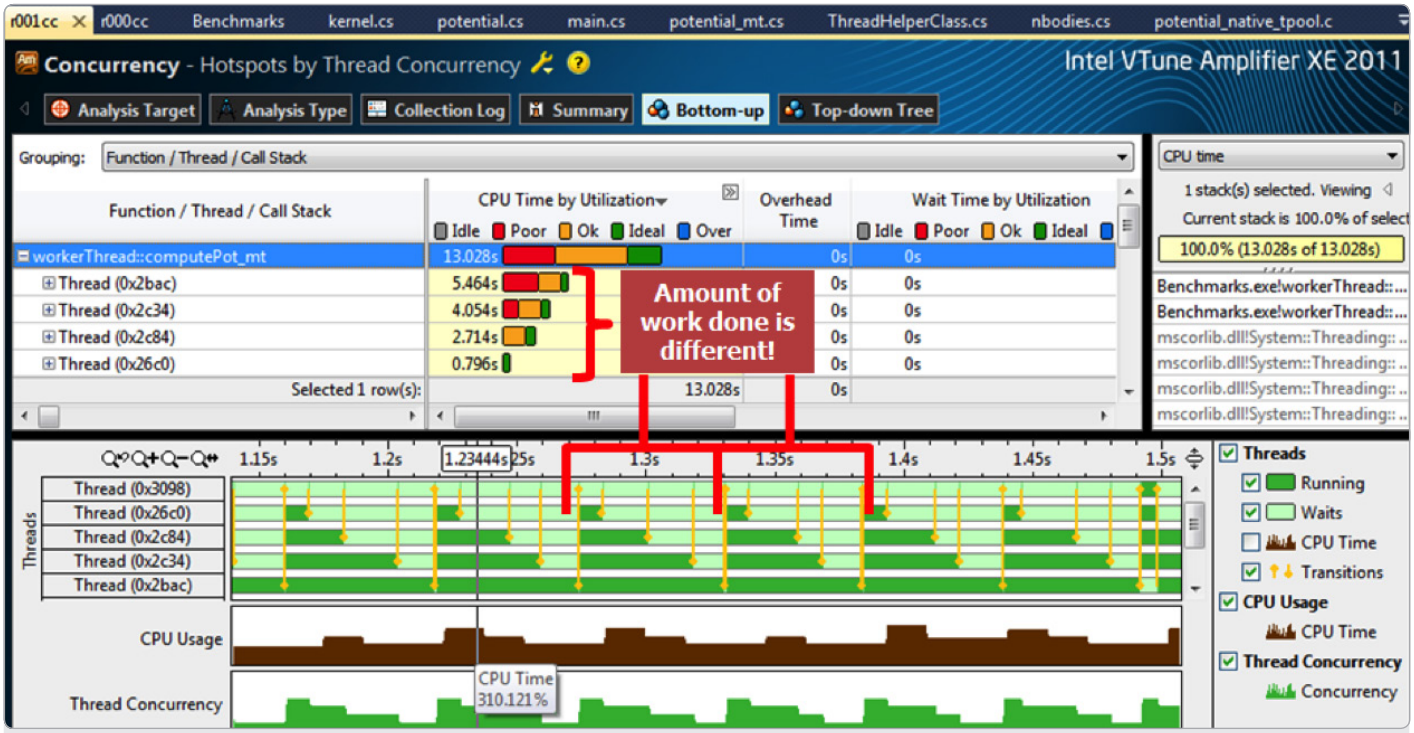


図 13: タイムライン上のスレッドのロード・インバランスと各実行スレッドの CPU 時間を表示

	スレッド 1（範囲）	スレッド 2（範囲）	スレッド 3（範囲）	スレッド 4（範囲）
オリジナルバージョン	0 -249	250 – 499	500 – 749	750 - 999
変更後	0,4,8,…,992, 996	1,5,9,..,993,997	2,6,10,…,994,998	3,7,11,…,995,999

図 14: 4 スレッドにおける反復の循環

オリジナルバージョン

```
chunkBegin = tid * (nParts / threadCount);
chunkEnd = (tid + 1) * (nParts / threadCount);
 …

start = threadParameters.chunkBegin;
end = threadParameters.chunkEnd;

for (int i = start; i < end; i++)
{
 for (int j = 0; j < i - 1; j++)
 {
    distx = Math.Pow((r[0][j] - r[0][i]), 2);
    disty = Math.Pow((r[1][j] - r[1][i]), 2);
    distz = Math.Pow((r[2][j] - r[2][i]), 2);
    dist = Math.Sqrt(distx + disty + distz);
    potential += 1.0 / dist;
 }
}
```

ロードバランスを含むバージョン

```
for (int i = tid; i < constants.POT_NPARTS; i +=
threadParameters.threadCount)
{
 for (int j = 0; j < i - 1; j++)
 {
    distx = Math.Pow((r[0][j] - r[0][i]), 2);
    disty = Math.Pow((r[1][j] - r[1][i]), 2);
    distz = Math.Pow((r[2][j] - r[2][i]), 2);
    dist = Math.Sqrt(distx + disty + distz);
    potential += 1.0 / dist;
 }
}
```

図 15: スレッドカウントによるロード・インバランスの改善

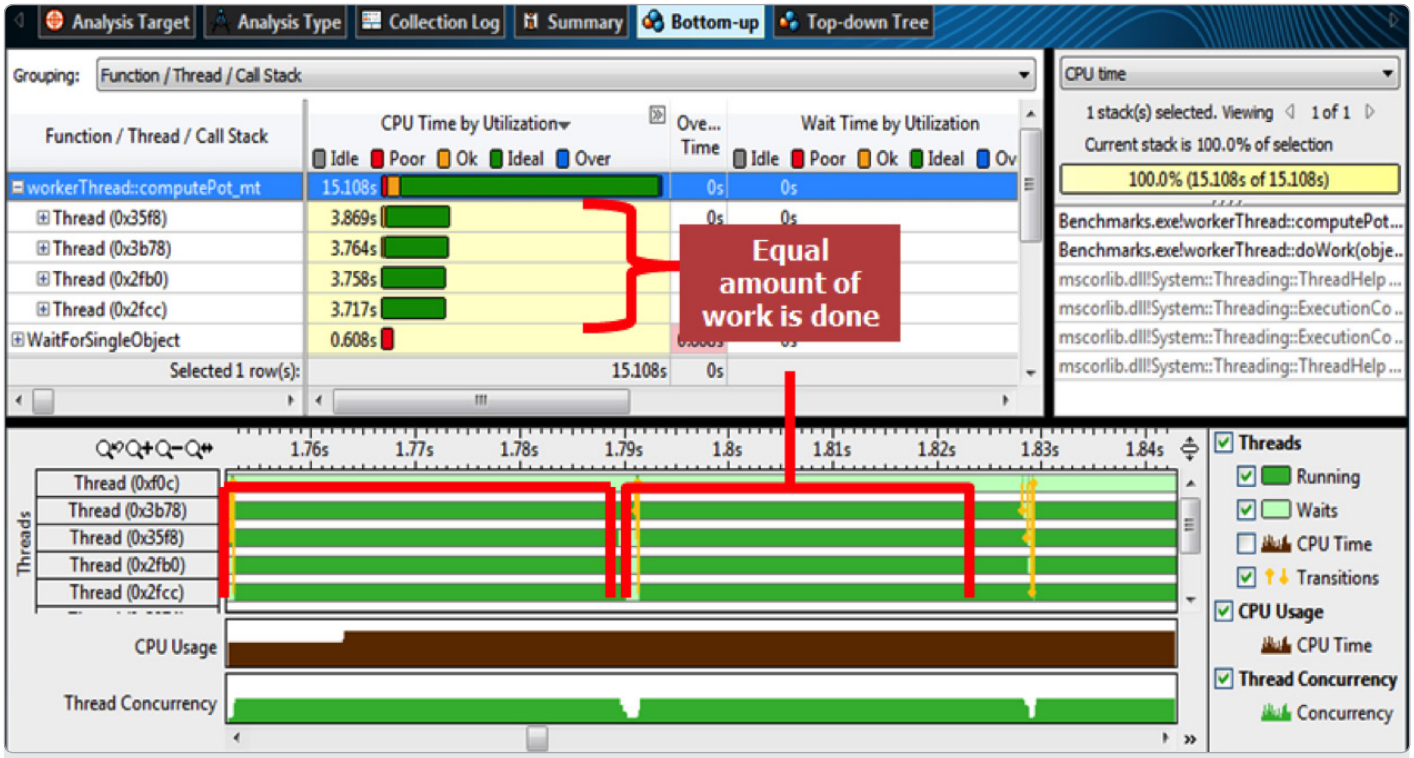


図 16: 作業がほぼ均等に分割された 4 つのスレッドで workerThread::computePot_mt メソッドが実行されている

これでコードの修正は完了でしょうか。いいえ、まだ続きがあります。インテル® VTune™ Amplifier XE のアーキテクチャー解析を実行して、さらなるパフォーマンス向上の可能性を探ってみます。ここでは、第 2 世代インテル® Core™ マイクロアーキテクチャー用に事前定義されている「General Exploration (全般解析)」を使用します。

第 2 世代インテル® Core™ マイクロアーキテクチャーは、理想的な状態では CPI が 0.25 まで低くなります。指定されたワークロードの CPI 値が大きくなるほど、コードをチューニングしてパフォーマンスを向上できる可能性がより高いことを意味します。図 18 は、[General Exploration (全般解析)] の結果を示しています。結果を見ると、`Math.Pow()` 関数の呼び出しに大量のクロック数が費やされていることが分かります。図 19 のように `Math.Pow()` を単純な乗算に置換することで、パフォーマンスが向上し、CPI レートは 1.5 に減りました。

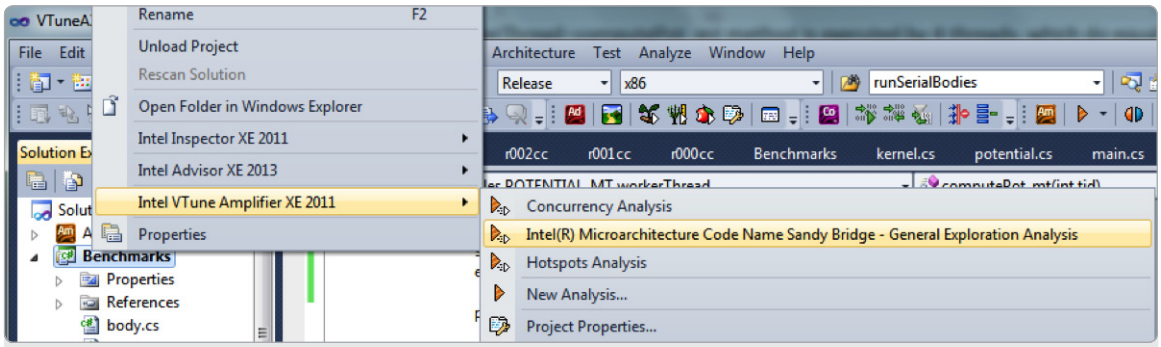


図 17: インテル® VTune™ Amplifier XE の [General Exploration (全般解析)] を実行

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

Grouping: Function

Function	Hardware Event Count		CPI Rate
	CPU_CLK_UNHALTED.THREAD	INST_RETIRED.ANY	
workerThread::computePot_mt	5,308,000,000	1,610,000,000	3.297
VTuneAmplifierXE::Examples::POTEI	4,000,000	2,000,000	2.000
workerThread::doWork	2,000,000	0	

図 18: [General Exploration (全般解析)] の結果 - 問題が潜む関数が強調表示されている

```
for (int i = tid; i < constants.POT_NPARTS; i +=
threadParameters.threadCount)
{
 for (int j = 0; j < i - 1; j++)
 {
    distx = Math.Pow((r[0][j] - r[0][i]), 2);
    disty = Math.Pow((r[1][j] - r[1][i]), 2);
    distz = Math.Pow((r[2][j] - r[2][i]), 2);
    dist = Math.Sqrt(distx + disty + distz);
     potential += 1.0 / dist;
 }
}
```

```
for (int i = tid; i < constants.POT_NPARTS; i +=
threadParameters.threadCount)
{
 for (int j = 0; j < i - 1; j++)
 {
   distx = (r[0][j] - r[0][i]) * (r[0][j] - r[0][i]);
   disty = (r[1][j] - r[1][i]) * (r[1][j] - r[1][i]);
   distz = (r[2][j] - r[2][i]) * (r[2][j] - r[2][i]);
   dist = Math.Sqrt(distx + disty + distz);
   potential += 1.0 / dist;
 }
}
```

図 19: Math.Pow() の置換

オリジナルバージョン (秒)	ロード・インバランス修正後 (秒)	Math.Pow() 置換後 (秒)
5.4	4.65	0.39

図 20: 実行時間の比較

「インテル® Parallel Studio XE を利用することで、.NET 開発者は、最も時間を費やしている関数やコード行などの重要なパフォーマンスの問題、スケーラビリティーの問題、同期構造や I/O アクティビティーの待機時間を特定することができます。インテル® Parallel Studio XE は、分岐予測ミス、キャッシュミス、メモリー帯域幅問題などによって引き起こされる潜在的なマイクロアーキテクチャーのボトルネックを発見します。」

パフォーマンス・チューニングの評価基準

リタイアした命令あたりのクロック数 (CPI) は、各命令の実行にかかったおおよその時間を示す基本的なパフォーマンスの評価基準です (単位はクロック数)。最近のスーパースカラー・プロセッサーは、1 サイクルあたり 4 つまでの命令を発行できるため、理論的な最良の CPI 値は 0.25 になります。さまざまな要因 (大きなレイテンシーのメモリー、浮動小数点演算、SIMD 演算、分岐予測ミスによりリタイアしない命令、フロントエンドの命令スタベーションなど) により、測定される CPI の値は高くなる傾向にありますが、アプリケーションのパフォーマンス・チューニングの全体的な可能性を判断する優れた測定基準として広く用いられています。

高度で詳細なマイクロアーキテクチャー解析を行えるように、ツールには定義済みの解析タイプが用意されています。パフォーマンス・モニタリング・ユニット (PMU) を使用してプロセッサー・イベントをサンプリングすることで、キャッシュミス、ストールサイクル、分岐予測ミス、その他のマイクロアーキテクチャーの問題を特定します。高度な解析タイプは、インテル® Core™2 マイクロアーキテクチャー、インテル® Core™ マイクロアーキテクチャー (開発コード名: Nehalem および Westmere) 、第 2 世代インテル® Core™ マイクロアーキテクチャー (開発コード名: Sandy Bridge) などのプロセッサー・アーキテクチャー向けに定義されています。これらの解析タイプを使用すると、ツールは問題の関数をハイライトしてヒントとアドバイスを表示します。

サポートされている .NET のバージョン

インテル® VTune™ Amplifier XE とインテル® Inspector XE は、.NET バージョン 2.0 から 3.5 で利用可能な基本的な同期メカニズムをサポートします。.NET 4.0 で採用された新しい同期 API や新しいタスク並列ライブラリ (TPL) はサポートしていません。

まとめ

エラー検出ツール インテル® Inspector XE とパフォーマンス分析ツール インテル® VTune™ Amplifier XE は、.NET 開発者にとって貴重なテクノロジーを提供します。インテル® Parallel Studio XE に含まれるこれらのツールを利用することで、アプリケーションのパフォーマンスを向上し、ハイパフォーマンス・コンピューティングとエンタープライズ・アプリケーションで必要とされるコードの品質と信頼性を高めることができます。このスイートには、パフォーマンス向上のために必要なすべてのツールが備わっており、将来のマルチコア・プロセッサーからメニーコア・プロセッサーへの移行も容易に行えます。

ソースコードのダウンロード ›

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

最適化に関する注意事項

インテル® コンパイラーは、互換マイクロプロセッサー向けには、インテル製マイクロプロセッサー向けと同等レベルの最適化が行われない可能性があります。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® SSE2）、インテル® ストリーミング SIMD 拡張命令 3（インテル® SSE3）、ストリーミング SIMD 拡張命令 3 補足命令（SSSE3）命令セットに関連する最適化およびその他の最適化が含まれます。インテルでは、インテル製ではないマイクロプロセッサーに対して、最適化の提供、機能、効果を保証していません。本製品のマイクロプロセッサー固有の最適化は、インテル製マイクロプロセッサーでの使用を目的としています。インテル® マイクロアーキテクチャーに非固有の特定の最適化は、インテル製マイクロプロセッサー向けに予約されています。この注意事項で対象としている特定の命令セットに関する詳細は、該当製品のユーザーズガイドまたはリファレンス・ガイドを参照してください。

改訂 #20110804

intel Software

パフォーマンスを極限まで高める

インテル® Parallel Studio XE

インテル® Parallel Studio XE 2011 は、個人ユーザーから数千人規模の大企業まで、あらゆる開発者に、かつてないアプリケーション・パフォーマンスと信頼性を提供する業界最高レベルの総合開発ツールです。下記の製品が含まれています。

高度な C/C++、Fortran コンパイラーとライブラリー
インテル® Composer XE

高度なスレッド / メモリーエラー検出、セキュリティー分析ツール
インテル® Inspector XE

高度なパフォーマンス分析ツール
インテル® VTune™ Amplifier XE

Rock your code. **Rock your world.**

インテル® Parallel Studio XE の評価版は、**www.intel.com/software/products**（英語）からダウンロードできます。製品情報および購入情報は、インテル® ソフトウェア開発製品 Web サイトを参照してください。
http://www.intel.com/jp/software/products/

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。
© 2012 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、Pentium、Xeon、VTune は、アメリカ合衆国および / またはその他の国における Intel Corporation の商標です。* その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。