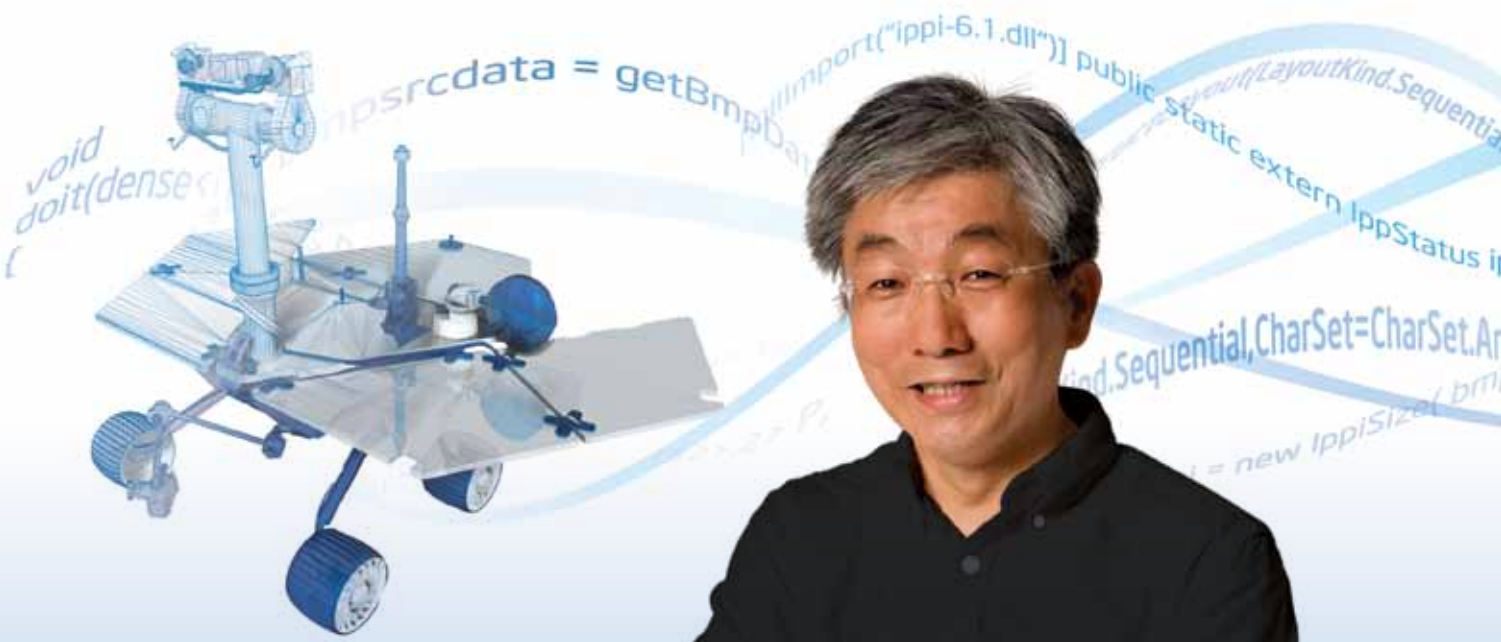
THE PARALLEL UNIVERSE

Issue 7
2011 年 6 月



スポットライト

池井満が最新技術への取り組みについて語ります。

インテル® Cilk™ Plus:
並列プログラミングのための C/C++ 言語拡張

Robert Geva

シリアル / 並列アプリケーションのパフォーマンス解析の簡素化

Levent Akyil

インテル® Parallel Building Blocks の 3 つの ‘for’ ループ

Noah Clemons

デベロッパー・ロックスター:
池井 満

目次

編集者からのメッセージ

各論 : FLOP カウントと並列プログラミング

JAMES REINDERS 3

インテル コーポレーションのソフトウェア開発製品部門のチーフ・エバンジェリスト兼ディレクターである James Reinders が、最もよく受け取る 3 つの質問 — インテル® VTune™ Amplifier XE で浮動小数点演算をカウントする方法、インテル® Cilk™ Plus とは、インテル® Parallel Building Blocks (インテル® PBB) — に答えます。

インテル® Cilk™ Plus : 並列プログラミングのための C/C++ 言語拡張

ROBERT GEVA 5

インテルは、インテル® Parallel Building Blocks (インテル® PBB) と呼ばれる新しいプログラミング・モデルに基づく並列プログラミング向けの新製品を提供しています。

インテル® Parallel Building Blocks (インテル® PBB) の 3 つの ‘for’ ループ

NOAH CLEMONS 11

インテル® Parallel Building Blocks の各モデルでは、異なる種類の ‘for’ ループを利用できます。モデルを選択する前に、ビルド環境、並列化の種類、抽象化のレベルを知っておくことがなぜ重要なのかを説明します。

シリアル / 並列アプリケーションのパフォーマンス解析の簡素化

LEVENT AKYIL 15

インテル® VTune™ Amplifier XE は、アプリケーションの問題を識別するための強力なパフォーマンス解析ツールです。操作性が向上した直感的なユーザー・インターフェイスにより、マウスを数回クリックするだけで強力なパフォーマンス解析を実行できます。

ケーススタディー : マサチューセッツ総合病院 *

BEVIN BRETT 25

マサチューセッツ総合病院とインテルの開発者が、シリアル・アルゴリズムを解析して C++ コードを変更し、並列化を行うことで仮想大腸内視鏡検査の処理を向上させた事例を紹介します。

©2011 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 Core、VTune は、アメリカ合衆国および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。

各論：FLOP カウントと並列プログラミング

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、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。

今回は、よく聞かれる 3 つの質問にお答えします。

1. インテル® VTune™ Amplifier XE を使用して浮動小数点演算をカウントする方法
2. インテル® Cilk™ Plus とは？
3. インテル® Parallel Building Blocks に用意されているさまざまな “parallel for” について

1. 浮動小数点演算

プログラムを実行した後、インテル® VTune™ Amplifier XE は「プログラムは 12,506,718,902 の浮動小数点演算を使用しました。」といったレポートを出力しますが、多くのユーザーはこの数字だけでは満足せず、「どのように浮動小数点演算を使用したのか？」、あるいは「どこで浮動小数点演算を使用したのか？」というように、より詳細な情報を知りたいと考えています。幸い、インテル® VTune™ Amplifier XE では、プログラムの正確なスレッドとステートメント行やスレッド / 関数レベルの総計を知ることができます。インテル® VTune™ Amplifier XE は統計サンプリングによって浮動小数点演算を推定するため、単一実行でシステムの動作に関する詳細情報を収集する間、ほとんど干渉しません。

パフォーマンスを追及するプログラマーは、レイテンシーの大きい浮動小数点演算がアプリケーションに与える影響を確認したいと考えるでしょう。除算や平方根を含む演算は実行に時間がかかるため、特に削除または縮小の対象になります。インテル® VTune™ Amplifier XE は、それらの演算がどこで、どのくらい行われているかを正確に示します。

浮動小数点演算には多くの種類があり、一部はストリーミング SIMD 拡張命令 (SSE) やアドバンスト・ベクトル・エクステンション (AVX) のようなベクトル命令を使用します。ベクトル命令は 1 つの命令で複数の浮動小数点演算を実行するため、ベクトル命令を使用することでパフォーマンスの向上につながります。インテル® VTune™ Amplifier XE で提供される詳細な情報により、ベクトル命令を使用した場合と使用しない場合でどの程度パフォーマンスが異なるかを判断することができます。

浮動小数点演算を多用するプログラムのパフォーマンスをチューニングする場合、今回の記事がインテル® VTune™ Amplifier XE を利用するきっかけとなるかもしれません。

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（英語）を参照してください。

「ベクトル命令は 1 つの命令で複数の浮動小数点演算を実行するため、ベクトル命令を使用することでパフォーマンスの向上につながります。」

2. インテル® Cilk™ Plus

Cilk は 1994 年に M.I.T. で最初に開発されました。現在提供されているインテル® Cilk™ Plus は、Windows*、Linux* をサポートしており、Mac OS* X のサポートも追加される予定です。インテル® Cilk™ Plus は、C/C++ プログラムに並列化を実装する非常に単純かつ強力な手法を提供します。この単純かつ強力な手法は、コンパイラー・テクノロジーにより実現されます。コンパイラーは、既存のプログラムに追加された非常に単純なキーワードを判断して、シリアル・アプリケーションを並列アプリケーションに変更します。

最初の Cilk には Cilk spawn と Cilk sync の 2 つのキーワードしかありませんでした。この単純性は万人に受け入れられるものでした。関数をバックグランド（別のスレッド）で実行する能力と、同期によりスポーンしたすべての関数を待つ柔軟性があれば、当初は十分でした。

インテル® Cilk™ Plus では、このほかに 3 つの項目が追加されています。最初の項目は、Cilk ‘for’ です。‘for’ を “cilk_for” に変更してループを並列ループに変換する手法は、非常に簡単に習得して利用できます。反復をそれぞれスポーンしないでループを並列化する場合、すべての反復をタスクにする必要があるため、オーバーヘッドが大きくなります。コンパイラーにループの反復が並列に行われるという情報を伝えることで、コンパイラーはランタイムに「最適な処理を行う」コードを生成することができます。プログラムをシングルコアで実行する場合に反復を複数のタスクに分割する必要性は、クアッドコア・システムで実行する場合とは異なります。

Cilk に追加された 2 つ目の項目はハイパーオブジェクト（レデューサー）で、今回の記事では説明していません。将来の記事で詳しく取り上げる予定です。レデューサーの詳細を知りたい場合は、コンパイラーのドキュメントを参照してください。レデューサーは、共有変数のプライベート・コピーを作成して、ボトルネックなしで共有変数を並列処理できるようにします。リダクション操作も可能です。難しく思えるかもしれませんが、レデューサーは単純であり、習得も簡単です。今回の記事では、レデューサーという機能が追加されているということを知っておいてください。

インテル® Cilk™ Plus に追加された 3 つ目の項目は、データ並列演算を明示的に指定する、C/C++ 向けの新しいベクトルと配列表記です。例として、以下のようなループの記述を考えてみましょう。

```
for (i=0; i<10000; i++) {
  a[i] = b[i] + c[i];
}
```

以下のコードを記述したい場合でも、これまでは上記のように記述する必要がありました。

```
a[:]= b[:]+ c[:];
```

インテル® Cilk™ Plus では、後者のコードをそのまま記述することができます。より複雑な配列セクションや要素関数の制御方法を含む、インテル® Cilk™ Plus に関する詳細な説明は、この後の記事で紹介します。

今後は、より多くのコンパイラーや規格でインテル® Cilk™ Plus の機能を目にする機会が増えることになるでしょう。

3. インテル® Parallel Building Blocks（インテル® PBB）のループ

今回の記事では、インテル® PBB でループを並列に処理する方法を説明します。そのバリエーションは少なく直感的なので、有益なだけでなく、強力な方法であることがお分かりになると思います。

記事のさまざまなトピックでは、よく聞かれる質問に対する答えを詳細に説明しています。皆様にとって必要な回答が得られることを期待しています。今後もご質問をお寄せください。

James Reinders
2011 年 6 月

Intel® Cilk™ Plus:

並列プログラミングのための C/C++ 言語拡張

インテル® ソフトウェア・サービス事業部
主任エンジニア
Robert Geva

インテルは、インテル® Parallel Building Blocks に基づく並列プログラミング向けの新製品を提供しています。ここでは、その中の重要なツールの 1 つであるインテル® Cilk™ Plus について説明します。

編集者からのメッセージ :

Robert は、Cilk の 3 つのキーワード（_Cilk_spawn、_Cilk_sync、_Cilk_for）を紹介します。次の記事の著者である Noah と私も含め、ほとんどの開発者はプログラムに cilk.h をインクルードして、cilk_spawn、cilk_sync、cilk_for の方を使用していますが、コンパイラーは有効な名前との衝突が発生しうる通常の名前空間ではこれらの新しいキーワードを作成しません。先頭に下線の付いたバージョンは実際のキーワードです。cilk.h インクルード・ファイルで定義されているバージョンの方が簡単ですが、Robert はあえて実際の名前の方を使用しています。どちらを選択するかはあなた次第です。

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（英語）を参照してください。

srcdata = getBmpData
[DllImport("ippi-6.1.dll")] public
bmpsrc.Width*3/4, bmpsrc.Heigh
std::co
2>,2> p
utKind.Seq

マルチコア・プロセッサーがサーバーだけでなく、デスクトップ、ノートパソコン、携帯端末に搭載されていることも珍しくなくなりました。インテル® プロセッサーでもマルチコアの占める割合がますます増加しており、アプリケーションのパフォーマンス、ユーザビリティー、消費電力、その他の向上に寄与しています。マルチコアシステム向けのコードを記述する開発者を支援するため、インテルは OpenMP* や MPI* などの規格への投資を続けています。同時に、インテルでは、**インテル® Parallel Building Blocks（インテル® PBB）と呼ばれる新しいプログラミング・モデルに基づく並列プログラミング向けの新製品を提供しています。インテル® PBB は現在、以下の 3 つのコンポーネントで構成されています。**

1. **インテル® Cilk™ Plus:** タスクとデータ並列構造の両方を提供する C/C++ 言語拡張です。現在は、**インテル ® C++ コンパイラー**で実装、サポートされています。この記事で紹介します。
2. **インテル® スレッディング・ビルディング・ブロック（インテル® TBB）**：インテル® TBB は、低レベルのタスク処理プリミティブと高レベルの並列アルゴリズムの両方を提供し、コンパイラーに依存しない C++ 向けのテンプレート・ライブラリーとして実装されます。サポート付きの製品とオープンソースがあります。
3. **インテル® Array Building Blocks（インテル® ArBB）**：インテル® ArBB は、特定の並列メカニズムやハードウェア・アーキテクチャーに依存しない、ベクトル並列プログラミングの汎用ソリューションを提供する C++ ライブラリーです。2011 年 8 月現在、ベータ版が提供されています。

プログラマーの関心

インテル® PBB のプログラミング・モデルは、プログラマーの観点と、プロセッサー・ハードウェアで提供される複数のレベルの並列リソースを考慮しています。

プロセッサー・アーキテクチャーは、マルチコア、ベクトル、ハイパースレッディング、キャッシュを含む、複数の並列リソースを提供します。プログラマーによって、それぞれのリソースの活用に対する興味の度合いは異なるでしょう。インテル® PBB（特にインテル® Cilk™ Plus）では、2 つのアプローチをサポートしています。1 つ目は、コアレベルで並列処理を明示的に記述する方法です。プログラマーは、タスクを使用して、各タスク内に明示的なベクトルレベルの並列処理を記述できます。2 つ目は、コンパイラーとシステム・ソフトウェアに処理を指示する方法です。このアプローチでは、並列に行われる処理が明確に定義され、処理するデータが明確に示されると仮定しています。

ハードウェア・リソースの活用に加えて、インテル® PBB（特にインテル® Cilk™ Plus）は、「相互運用性」を実装します。多くのアプリケーションは、独自にまたはサードパーティーによって作成された（ライブラリーなど）、独立したコンポーネントで構成されています。一般に、コンポーネント（とその設計者）は、低レベルではなく、高レベルのインターフェイスを利用して互いに通信します。並列プログラムでそのようなレベルでの共同作業を可能にするには、タスク・スケジューラーが相互運用性をサポートする必要があります（つまり、モジュールを組み合わせてアプリケーションにしたときに、各モジュールが継続して正しく動作する必要があります）。

インテル® Cilk™ Plus はどのように相互運用性をサポートしているのでしょうか？インテル® Cilk™ Plus 言語拡張は、インテル® PBB のほかのコンポーネントと同様に、ユーザーモードのワークスチール・タスク・スケジューラーを利用します。ワークスチールは、ほかのワーカーに処理する作業がない場合、作業を生成したワーカーからほかのワーカーに作業を移行（スチール）して実行できる、タスクスケジューリング・テクニックです。作業の実行を完了したワーカーは、ほかのワーカーのキューにある作業アイテムをスチールします。しかし、並列処理可能な作業を所有しているワーカーがほかのワーカーにその作業を委任することはできません。この記事では、言語の面からスケジューリングについて取り上げますが、そのメカニズムの詳細な説明はしません。ワークスチール・テクニックは、並列プログラムのモジュール実装を可能にする、相互運用性の中核となるものです。

インテル® Cilk™ Plus によるタスク並列化

インテル® Cilk™ Plus 言語拡張はタスクレベルの並列化のために 2 つの構造を提供します。1 つ目の構造には、_Cilk_spawn と _Cilk_sync の 2 つのキーワードが含まれます。図 2 および図 3 は、有名なフィボナッチ数列の再帰呼び出しのシリアル C コードと、新しいキーワードを追加してシリアルコードから派生した並列実装の比較です。

インテル® Parallel Building Blocks

優れたアプリケーション・パフォーマンスを提供する並列プログラム開発モデルの総合セット

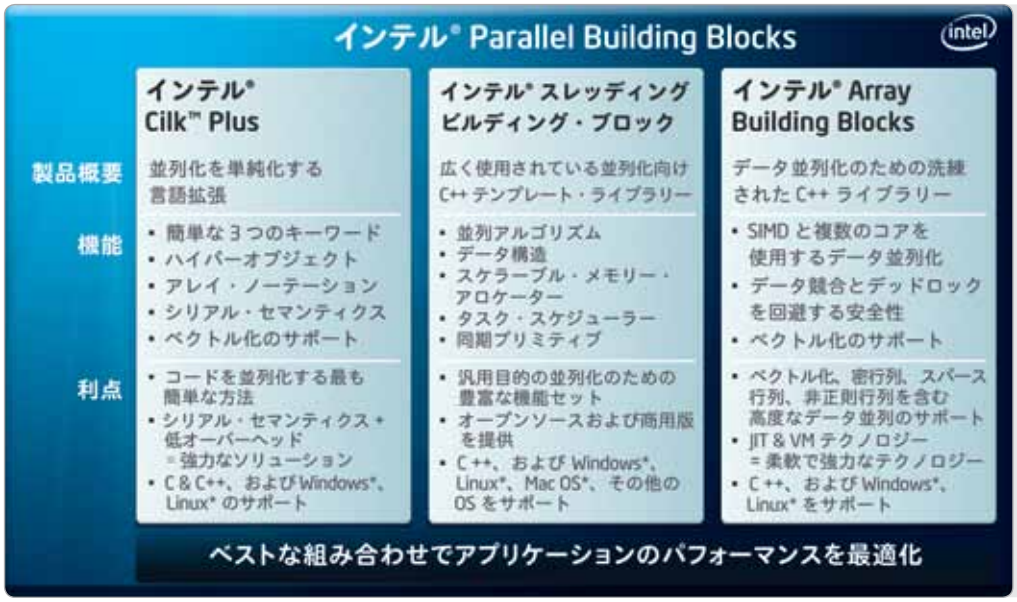


図 1: インテル® Parallel Building Blocks のサマリー

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（英語）を参照してください。

シリアルコード

```
int fib (int n)
{
  if (n <= 2)
      return n;
  else {
      int x,y;
      x = fib(n-1);
      y = fib(n-2);
      return x+y;
  }
}
```

図 2: フィボナッチ数列のシリアル実装

インテル® Cilk™ Plus のキーワードで並列化されたシリアルコード

```
int fib (int n)
{
  if (n <= 2)
      return n;
  else {
      int x,y;
      x = _Cilk_spawn fib(n-1);
      y = fib(n-2);
      _Cilk_sync;
      return x+y;
  }
}
```

図 3: インテル® Cilk™ Plus のキーワードを使用して並列化された同じコード。インテル® Cilk™ Plus のキーワードが挿入されているだけで、元のコードは何も変更されていないことに注意。

キーワードの追加はプログラムに干渉しません。シリアルプログラムから並列プログラムへの変更に必要なのは、キーワードを適切に挿入することだけです。プログラムそのものを変更する必要はありません。キーワードの挿入がプログラムの可読性や保守性に影響することもありません。元のシリアルコードは簡単に確認でき、プログラムのロジックは明白です。これは、インテル® Cilk™ Plus が、その設計原理の 1 つである、既存のシリアル C/C++ コードを並列化するように設計されているためです。そのため、インテル® Cilk™ Plus を利用することで、プログラマーは、シリアルコードとの同等性を保ちながら、最小限の変更でコードに並列化を簡単に実装できるようになります。

_Cilk_spawn キーワード

では、_Cilk_spawn キーワードについて詳しく見ていきましょう。このキーワードは、図 3 の fib(n-1) のような関数呼び出しに適用します。これは、スポーンされた関数がその関数の残りの部分（この例では fib(n)）と同時に実行可能であることを意味します。_Cilk_spawn が適用されている関数は「スポーン関数」または親と呼ばれます（スポーンされる関数が子）。_Cilk_sync キーワードは、子タスクが親に制御を返すまで親が待たなければならないことを意味します。

このメカニズムは、並列領域（親側）とスポーンされるユニット（子側）で（C/C++ プログラミングでよく知られている）関数の概念を再利用することに注意してください。どちらも、最小限の労力で既存のプログラムに並列化を実装することが目的です。並列領域として再利用することで、並列領域のために新しい構文を導入する必要性が軽減されます。スポーン単位として関数を再利用することは、制限と見なされるかもしれません。既存のプログラムに並列化を実装するとき、プログラマーは、ステートメントやループなどのより小さなレキシカルスコープが、ほかのステートメントと同時に実行できることを示そうとします。

しかし、スポーン単位として関数を使用することで、データモデルはプログラマーにそのまま理解されます。また、言語拡張にプログラミング・エラーの原因となる新しい規則を追加する必要がありません。現在のモデルでは、スポーン関数のスコープ内にある変数、子関数に属している変数、引数がどのように渡されるかがプログラマーに明らかです。引数の渡し方は基本の言語で行われる方法に準拠しています。引数の値は、子を分離して並列化を有効にする前に、親タスクによって評価されます。図 3 の fib(n-1) の例では、n-1 の値は子がスポーンされる前に親によって計算されます。より複雑な、引数の計算がより多く含まれている例を図 4 に示します。

図 4 の関数 f() の引数は、それ自身が関数呼び出しです。_Cilk_spawn の操作が式全体に適用されると、g() と h() は、互いの関数とスポーン関数に注意を払いながら同時に実行できるようになります。関数 g() と h() 内のコード間でデータ競合が存在するケースでは、同期宣言子を挿入する場所はありません。

おそらく、_Cilk_spawn の動作で最も重要な属性であるシリアル実行との同等性のサポートは、_Cilk_spawn ポイントで親の 2 つの部分が非対称であることに関係があります。 _Cilk_spawn は並列実行を強制するわけではなく、並列実行の機会を提供します。_Cilk_spawn キーワードに到達したワーカーは、キーワードによって識別された 2 つの分岐の 1 つを処理し続けます。残りの分岐は分離され、後の実行のためにキューに入れられます。前述したように、キューの作業アイテムは別のコアで実行している別のワーカーによってスチールされる場合もあれば、されない場合もあります。

```
val = _Cilk_spawn f(g(x),h(y));
```

図 4: より複雑な _Cilk_spawn の例。

直感的には、スポーンされた関数が後の実行のためにキューに入れられ、親を実行しているワーカーは実行を続けると思われがちです。しかし、実際はその逆です。_Cilk_spawn キーワードに達したワーカーは、作業の実行を続けるために分離され、キューに入れられます。

なぜこの非対称性が重要なのでしょうか。その理由は、評価の順序が、シリアル C/C++ プログラムの関数呼び出しの評価の順序に一致しているためです。継続処理が別のワーカーによってスチールされなければ、スポーン関数は、通常の関数呼び出しと同じように、子の実行を続けます。そして、リターンしたら、作業アイテムとして継続処理をポップし、その実行を続けます。ここで重要な点は、この評価の順序が、シリアルプログラムの関数呼び出しの評価の順序と同じであるということです。

インテル ® Cilk ™ Plus のキーワードは無効にできるということも重要です。例えば、プログラマーは、図 5 の構文を使用し、C/C++ プリプロセッサーと #define を使用してキーワードを無効にすることができます。

```
#define _Cilk_spawn
#define _Cilk_sync
```

図 5: インテル® Cilk™ Plus のキーワードは無効にできるので、インテル® Cilk™ Plus のキーワードをサポートしないコンパイラーでビルドされたアプリケーションは、シリアル・アプリケーションとして正しく実行できます。このため、ソースコードの移植性が保たれます。

インテル® Cilk™ Plus のキーワードは無効にできるので、これらのキーワードを使用するアプリケーションをコンパイル / ビルドすると、有効なシリアル・アプリケーションが生成されます。プログラムの並列バージョンでデータ競合が発生しない限り、プログラムのシリアルバージョンと並列バージョンは同等で、同じ結果になります。

_Cilk_sync キーワード

図 3 の fib() 関数の例で、変数 x と y の値が使用される前に fib(n-1) と fib(n-2) の非対称な実行により更新されるようにするには、_Cilk_sync キーワードが必要です。キーワードによる明示的な指定に加えて、すべてのスポーン関数の最後には暗黙の _Cilk_sync があります。暗黙のキーワードは、実際にコンパイラーによって挿入されます。

暗黙の _Cilk_sync により、スポーン関数は、スポーンされた子のいずれかが実行されている限り実行されます。この特性の 1 つの利点は、プログラマーが作業単位として関数を見続けられること、そして関数がいつすべての作業をリターンし、完了したのか確認できることです。より実用的な利点は、スポーン関数がスタック変数のいずれかのアドレスを子に渡しており、子タスクがその場所に書き込みを行う場合、子の実行中に親のスタックがメモリーにあることが保証されることです。

_Cilk_for キーワード

2 つ目の構造には、_Cilk_for キーワードが含まれます。このキーワードを利用すると、プログラマーは C/C++ の for ループを並列化することができます。図 6 のステートメントを見てみましょう。

このキーワードが使用されると、コンパイラーは ‘for’ 構文に以下の制限を加えます。

```
_Cilk_for (int i = 0; i < N; ++i) {
  body;
}
```

図 6: _Cilk_for キーワード

- ループの 3 つの式すべてのインデックス変数 (図 6 の例では i) は、最初の式で初期化されます。
- インデックス変数はループ内で変更されない値と比較されます。
- インデックス変数はループの内部で変更されない値によってインクリメントされます。
- インデックス変数はループ本体では変更できません。

これらの制限の影響により、ループが実行されるとき、そのトリップカウントがランタイム・スケジューラーに通知されます。

_Cilk_for ループの実行には、分割統治アプローチが使用されます。実行が _Cilk_for キーワードに達したワーカーは、トリップカウントを計算します。後の評価のため、反復の上半分を自身のワークキューに入れます。トリップカウントが十分小さくなるまで (ランタイムでヒューリスティックに決定)、より小さなトリップカウントで分割統治を続けます。その後、反復を連続して実行します。実行が完了すると、ワーカーは自身のワークキューから、最後にポストされた次の作業アイテムをポップします。これが下半分の反復の 2 つ目のセットになります。別のワーカーのワークキューが空の場合、そのワーカーはループ反復に対応する作業アイテムのリストを作成したコアから作業をスチールします。これで、同時に行われる作業が作成されます。

この場合、別のワーカーは反復の最大数に対応するキューのトップからアイテムをスチールします。この順序のワークスチールでは、アイテムの取得元であるワーカーのキャッシュの局所性に影響する可能性は最も低くなります。また、最も少ないスチール数でワーカーに最も多い作業を提供します。最小のスチール数を使用することで、最小のオーバーヘッドですべてのワーカーに作業を提供できます。作業がスチールされない限り、単一のワーカーによって実行されたループ反復の評価の順序は、シリアル実行の順序と全く同じです。

インテル® Cilk™ Plus のベクトルレベルの並列化

インテル® Cilk™ Plus 言語拡張は、インテル® マルチコア・プロセッサーで利用可能なハードウェア・ベースの並列化を活用するいくつかの方法を提供します。

シリアル ‘for’ ループは、キーワードを ‘for’ から “_Cilk_for” に変更することで並列ループに変更できます。この変更は、ループの反復に順序付けがないことをコンパイラーに伝えます。コンパイラーは、並列に実行する反復のバッチを用意して、連続する反復のバッチ内のコードのベクトル化を試みます。

配列表記 (アレイ・ノーテーション)

インテル® Cilk™ Plus で提供される新しい構文は、配列に対する単純な操作を可能にします。C/C++ 言語標準には配列の操作を表現する方法はありません。代わりに、プログラマーがシリアル順にループを記述し、配列の要素の点から操作を表現しなければなりません。シリアル順にループを記述するマイナス面は、シリアルループをベクトルループに変換するために、ベクトル処理がループに起因し、その言語で要求されるスカラー処理と等価であることをコンパイラーが証明しなければならないことです。ほとんどの場合、この証明は失敗します。

例えば、プログラムで配列を参照するポインターを使用している場合 (特定の配列を操作するためのコードではなく、任意の配列の操作を考慮するプログラミングにおいて)、コンパイラーは 2 つのポインターが、交差しないメモリー領域を指していることを証明できないでしょう。

a[:] = b[:] + c[:]; という記述は、配列 “b” と “c” の要素を追加する必要があり、追加操作に決められた順序がないことをコンパイラーに通知します。その結果、コンパイラーは、スカラーコードを生成してベクトルコードと等価であることを証明しようとする代わりに、常にベクトルコードを生成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 (英語) を参照してください。

インテル® Cilk™ Plus の配列表記演算子

インテル® Cilk™ Plus 言語拡張は、構文が array_base[start:length:stride] の配列表記演算子を定義します。

- array_base は、配列、ポインター、C99 可変長配列を含む、C/C++ で許可される任意の配列式です。
- start は、最初の配列要素です。
- length は、配列要素の数です。
- stride は、要素間のインクリメントです。stride の指定はオプションです。指定しない場合、デフォルト値の 1 が使用されます。

表記演算子は多次元配列に適用でき、a[0:5][3][0:8:2] のように配列添字と組み合わせることができます。この例で、“a” は 3 次元配列または配列へのポインターでなければなりません。式の次元数は、添字ではなく、表記演算子が使用される次元の数です。この例では、次元数は 2 です。

配列の要素の合計のような、一般に使用される操作のために、いくつかの組み込み関数が提供されています。内積計算は次のように表現されます。

```
x = __sec_reduce_add(a[:] * b[:]);
```

図 7: __sec_reduce_add を使用した内積計算の表現

配列表記法は Fortran 90 の配列に似ているように見えるでしょう。大きな違いは、インテル ® Cilk ™ Plus の配列表記法は代入文の右辺の配列が左辺の配列と部分的にオーバーラップしないと想定していることです。この想定に反している場合、予期できない動作が発生します。このため、Fortran 90 とは異なり、コンパイラーは配列式の右辺の中間値を保持する一時配列を生成しません。これにより、高いパフォーマンスが実現されます。

配列表記は、プログラマーが配列操作レベルでアルゴリズムとデザインを表現する際に、プログラムを記述する自然な方法です。既存のシリアルプログラムでは、表記法の変更はプログラムに干渉することになります。インテル ® Cilk ™ Plus では、#pragma simd 形式の干渉の少ない変更も利用できます。ループにプラグマを追加することは、ループがスカラー構文で記述されていても暗黙のシリアル順序は意図されていないことを示します。プログラマーの意図は、コンパイラーがベクトルコードを生成してループを実装することです。(パフォーマンスのヒントではなく) 言語構造としてプラグマを使用すると、コンパイラーは、ベクトルコードがスカラーコードと等価であることと、スカラーコードがプログラマーの意図ではないことを証明する必要なしに、ベクトルコードを生成できます。

3 つ目のプログラミング構造は、プログラマーが標準 C/C++ でスカラー関数を記述して、それを「要素関数」として宣言できるようにします。要素関数を使用する場合、プログラマーの意図は、配列要素間の操作の順序を決めずに配列の多くの要素でその関数を展開することです。図 8 は、要素関数を使用して配列を要素単位で加算する単純な例を示しています。

__declspec(vector) の使用は、関数 “v_add” が要素関数として使用されるように意図されていることをコンパイラーに示します。インテル® C++ コンパイラーは、そのような関数について、標準バージョンとベクトルバージョン (x1 の引数のベクトルと x2 の引数のベクトルを受け取る) の 2 つのコードを生成します。ベクトルバージョンは、すべての入力引数にまたがるベクトルレジスターを使用して関数の操作を行い、単一の結果の代わりに結果のベクトルを返します。関数は、スカラー・コンテキストで呼び出すことができます。そのような状況では、コンパイラーは関数呼び出しを標準のスカラー関数の呼び出しに変換します。

```
__declspec(vector)
float v_add(float x1, float x2)
{
  return x1+x2;
}
caller:
for (int j = 0; j < N; ++j) {
  res[j] = v_add(a[j],b[j]);
}
```

図 8: 要素関数を使用して配列を要素単位で加算する例。

図 8 のループの例のように、関数がデータ並列コンテキストで呼び出された場合、コンパイラーはベクトルバージョンを呼び出します。コンパイルのターゲットが、例えば SSE 拡張命令セットと XMM ベクトルレジスターをサポートするプロセッサーの場合、コンパイラーは入力配列の連続する 4 つの値がレジスター内に収まるかどうかを判断します。そして、コンパイラーは連続する 4 つの配列要素を操作する関数を生成し、その関数は N/4 回呼び出されます (N が 4 の倍数でない場合、スカラーバージョンで 1 ～ 3 回追加)。その利点はもちろん、パフォーマンスの向上です。

プログラマーは、‘for’ ループを使用して関数を呼び出す代わりに、“_Cilk_for” ループを使用して関数を呼び出してもかまいません。その場合、単にコンパイラーがベクトルバージョンを呼び出すだけでなく、複数のコアを使用することでループ反復の実行が促進されます。この言語構造を使用することで、プログラマーはコアレベルの並列化とベクトルレベルの並列化の両方を組み合わせた利点を生かすことができます。

まとめ

インテル® Cilk™ Plus は、C/C++ の言語拡張です。以下のような利点があります。

1. アプリケーション開発者は、コア、ベクトル、キャッシュを含む、ハードウェアで利用可能な並列リソースをすべて使用することができます。
2. 開発者が並列に作業を行う方法 (ベクトルで行われる作業とコアで行われる作業) を選択できるように、さまざまなレベルの抽象化を提供します。または、特定のハードウェア向けに明示的にプログラムしないように選択して、代わりに並列化の意図を示し、コンパイラーがハードウェアに操作をマップできるようにします。
3. 相互運用性をサポートします。タスクとワークスチール・スケジューラーを使用することで、コンポーネント間の並列化アーキテクチャーを調整する必要なく、また統合によるハードウェア・リソースのオーバーサブスクリプションのリスクを心配することなく、個別に開発されたコンポーネント (サードパーティーのライブラリーやサブグループによって個別に開発されたモジュール) とアプリケーションを統合することができます。
4. 既存のシリアルプログラムの並列化を単純化します。拡張のさまざまな属性により、干渉しない構文、容易にシリアルプログラムに戻すことができる機能、シリアル・セマンティクスとの同等性の保証、オーバーヘッドの低いタスクのスポーンをサポートします。

3

インテル® Parallel Build
3 つの ‘FOR

Noah Clemons

インテル® Parallel Building Blocks の各モデルでは、異なる種類の ‘for’ ループを利用できます。モデルを選択する前に知っておくべきことを説明します。

一般的なパフォーマンスの課題は、処理時間のほとんどが 1 つ以上の ‘for’ ループで費やされる場合です。通常、プログラマーは最初に利用可能なコアの一部またはすべてで ‘for’ を並列化しようとします。**インテル® Parallel Building Blocks（インテル® PBB）に含まれている各モデルでは、以下の ‘for’ ループを利用できます。それぞれ類似点と相違点があります。**

- インテル® Cilk™ Plus : cilk_for
- 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）: parallel_for
- インテル® Array Building Blocks（インテル® ArBB):_for と arbb::map() の併用

モデルを選択する前に、それぞれの ‘for’ のビルド環境、表現される並列化の種類、提供される抽象化のレベルを理解しておくことが重要です。これらの情報を理解しておくことで、プログラマーは、リファレンス・マニュアルを読むことなく、どの ‘for’ を使用すべきか決定することができます。

インテル® PBB の ‘for’ ループは以下からなります。

- データ並列と汎用並列化ソリューション
- 言語拡張とテンプレート・ライブラリー・ソリューション
- 初級から上級までさまざまなレベルの API 制御

	データ並列かタスク並列か？	言語拡張かテンプレート・ライブラリーか？	API 制御
cilk_for	タスク並列 - ループ内部で配列表記法を使用してデータ並列性を実装	言語拡張	初級
parallel_for	タスク並列	テンプレート・ライブラリー	中級、ただし上級オプションを含む
_for + arbb::map()	データ並列	テンプレート・ライブラリー	中級

表 1

```
cilk_for (int x = 0; x < 1000000; ++x) { … }
```

図 1

```
int x = 0;
cilk_for(int i = 0; i < N; i++)
     x++;
```

図 2

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（英語）を参照してください。

ling Blocks（インテル® PBB）の
’ ループ

cilk_for

インテル® Cilk™ Plus の Cilk コンポーネントには、図 1 に示すように、‘for’ を実行するスレッドをスポーンするキーワード形式の構文があります。

このループの使用には多少の注意が必要です。

- 一部またはすべての反復が互いに並列に実行されます。
- プログラムが継続する前にすべての反復が完了します。

以下のような制約もあります。

- プログラマーが使用できる制御変数は 1 つです。
- 反復の開始地点へのジャンプはランダムに発生することをプログラマーは留意する必要があります。
- 反復は互いに独立している必要があります。

cilk_for は、parallel_for やインテル® ArBB の _for と比較すると、制御性は最も限られています。プログラマーは、インテル® Parallel Composer にスレッド化の判断をまかせることになります。図 2 は、変数をインクリメントしているだけですが、処理は並列に行われています。

cilk_for の使用は、‘for’ の作業を異なるプロセッサー・コアで同時に実行できるチャンクに分割できることをランタイムに示します。キーワードを使用するだけでは暗黙のベクトル化は行われません。ベクトル化には、インテル® Cilk™ Plus の配列表記コンポーネントを使用することを推奨します。cilk_for は単純なキーワードなので、ループのパラメーターやスレッド化の方法を操作するための柔軟性はあまりありません。しかし、それゆえに、cilk_for は習得とコーディングが非常に簡単であり、インテル® ツールで既存の ‘for’ ループを並列化する最も単純な方法となっています。

あるスレッドが x の値をロードしてインクリメントする前に、別のスレッドが x を読み込み、（インクリメントしないで）メモリーに戻す可能性があるため、x のインクリメントには潜在的なデータ競合問題があることに注意してください。データ競合を解決する方法の 1 つは、図 3 に示すように、インテル® Cilk™ Plus のレデューサーを使用することです。

提供されるインテル® Cilk™ Plus のテンプレート・クラス・タイプで変数を宣言するだけで、スケーラブルなパフォーマンスを維持しながら、共有変数を安全にインクリメントすることができます。set_value および get_value レデューサー・メンバー関数を使用すると、レデューサーにある並列領域外のデータにアクセスすることができます。

上記の単純な例で cilk_for が使用されていなければ、コンパイラーは ‘for’ ループをなくすように最適化することに注意してください。cilk_for をベクトル化と併用すると便利なこともよくあります。以下の例は、元の実装に 3 行のコードを追加するだけで、ベクトルの内積の実装をブロック化、ベクトル化、スレッド化しています。値を合計する際に発生する可能性のある競合状態を回避するために、図 4 ではレデューサーを使用していることに注意してください。

```
#include <cilk\reducer _ opadd.h>
cilk::reducer _ opadd<int> x;
x.set _ value(0);
cilk _ for(int i = 0; i < N; i++)
      x++;
// Use x.get _ value() to access x afterward
```

図 3

parallel_for

インテル ® TBB の parallel_for は、タスク並列アルゴリズムを実装する高い柔軟性を備えたテンプレート・ライブラリーを利用できるビルド環境に適しています。すでに並列プログラミングの経験があるプログラマーには、非常に便利です。並列プログラミングの習得コースやオンザジョブ・トレーニングで学ぶほとんどの概念は、API で簡単に使用できるようテンプレート化されています。

parallel_for は、舞台裏の詳細を操作する優れた機能を提供します。インテル® TBB は、スレッドの管理ではなくタスクの処理に焦点が当てられていますが、柔軟な処理を行うこともできます。粒度を自動的に決定したり、きめ細かな調整を行うオプションを明示的に指定できます。この制御は、図 5 の例を見るとよく分かります。

```
float sprod(float *a, float *b, int size) {
  float sum = 0.0;
  for (int i=0; I < size; i++)
    sum += a[i] * b[i];
  return sum;
}
```

```
float sprod(float* a, float* b, int size) {
 int s = 4096;
 cilk::reducer_opadd<float> sum(0);
 cilk_for (int i=0; i<size; i+=s) {
   int m = std::min(s,size-i);
   sum += __sec_reduce_add(
     a[i:m] * b[i:m] );
 }
 return sum.get_value();
}
```

図 4

インテル® TBB で並列の‘for’ループを構築するためにテンプレート化されたコンポーネントを使用していることに注意してください。

- ChangeArray クラスは parallel_for の for ループの本体を定義します。
- インテル® TBB のテンプレート blocked_range は 1 次元反復空間を表します。
- C++ の関数オブジェクトと同様に、メインの作業は operator() 内部で行われます。
- 最後に、ChangeArrayParallel を呼び出して、テンプレート関数 parallel_for<Range, Body>（引数 Range -> blocked_range Body -> ChangeArray）を呼び出します。

多くの場合、プログラマーはこれらのコンポーネントの一部を使用して for 形式の計算を行うことができますが、図 6 の blocked_range を使用したベクトル内積の例で示されているように、必ず‘for’を使用する必要はありません。

```
#include "tbb/blocked_range.h"
#include "tbb/parallel_for.h"
using namespace tbb;
class ChangeArray{
    int* array;
public:
    ChangeArray (int* a): array(a) {}
    void operator()( const blocked_
range<int>& r ) const{
        for (int i=r.begin(); i!=r.end();
i++ ){
            foo (array[i]);
        }
    }
};

void ChangeArrayParallel (int* a, int n )
{
    parallel_for (blocked_range<int>(0, n),
ChangeArray(a));
}
int main (){
    int A[N];
    // initialize array here…
    ChangeArrayParallel (A, N);
    return 0;
}
```

図 5

インテル® ArBB の _for

インテル® ArBB は、インテル® TBB のデータ並列に類似しています。スレッド化にインテル ® TBB を使用して、独自のランタイムコード生成プロセスによりベクトルコードを生成します。プログラマーは‘for’ループの計算について、ほかのインテル® PBB のモデルとは異なった考え方をします。インテル® ArBB は任意の計算を指定できるライブラリーおよび言語であり、キーワードは**任意**で、インテル® TBB よりもさらに高レベルです。Cilk の 3 つのキーワードと比較すると、インテル® ArBB は、プログラミング・プラットフォームそのものといえるでしょう。タスクの管理やベクトル化の細かい処理は任せてしまいたいアルゴリズム設計者に適しています。

インテル® ArBB の _for は C/C++ のループに似ていますが、単体で並列実行される訳ではなく、通常のシリアルループです。API 固有のほかの並列演算を使用するか、_for ループと arbb::map() を組み合わせて使用する必要があります。_for と arbb::map を併せて使用すると、プログラマーは、タスクを管理することなく、ベクトル化とスレッド化を行うことができます。

どのくらいの並列演算が抽象化され、‘for’ループが不必要になるか、図 7 の、動作はするものの実際に並列化されていない行列 / ベクトル乗算のインテル® ArBB 実装を確認してください。

インテル® ArBB の型、演算子、キーワードを使用していることを除けば、このコードはシリアル C 実装に非常に似ています。このコードは、C バージョンのようにシリアルに実行されます。これはインテル® ArBB を使用して効率的な並列プログラムを作成するための正しい方法ではありません。

インテル® ArBB 構文の単純性を示し、アルゴリズム固有の並列性を表現している実装については、図 8 を参照してください。

集合演算子の使用に加えて、スカラー演算子の代わりにコンテナー演算子が使用されていることが分かります。コードがより単純になっているだけではなく、インテル® ArBB のランタイムがベクトル化とマルチスレッド化により計算を並列化しています。

では、_for ループはいつ使用すれば良いのでしょうか？

_for ループは、以下の状況で使用するべきです。

- インテル® ArBB 関数の内部
- シリアルに依存する反復計算の表現。計算は段階的に行われ、現在のステップは前のステップの結果に依存します。図 9 で示されている反復ステンシルを使用した熱放散は良い例の 1 つです。

このコードでは、ステンシルベースの更新ステップの計算はそれぞれ arbb::map() 関数を使用して並列化されていますが、ソリューションを計算するために複数回の更新が、順に行われなければなりません。

インテル® PBB で表現される‘for’ループはそれぞれ、プログラマーの異なるニーズに焦点を当てています。ニーズに合わせて、最適なものを選択してください。o

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（英語）を参照してください。

```
float sprod(float *a, float *b,
            int size) {
  float sum = 0.0;
  for (int i=0; i < size; i++)
    sum += a[i] * b[i];
  return sum;
}
```

```
float sprod(const float a[], const float b[],
size_t n ) {
  return tbb::parallel_reduce(
    tbb::blocked_range<size_t>(0,n),
    0.0f,

    [=](
      tbb::blocked_range<size_t>& r,
      float in
    ) {
      return std::inner_product(
        a+r.begin(), a+r.end(),
        b+r.begin(), in );
      },
      std::plus<float>()
  );
}
```

図 6

```
void matvec_product(const dense<f32,
2>& matrix, const dense<f32>& vector,
dense<f32>& result)
{
    usize rows = matrix.num_rows();
    usize cols = matrix.num_cols();
    _for (usize i = 0, i < rows, ++i) {  //
SERIAL LOOP
        f32 sum(0.0);
        _for (usize j = 0, j < cols, ++j) {
// SERIAL LOOP
            sum += matrix(j,
i) * vector[j];

        } _end_for
        result = replace(result, i, sum);
    } _end_for
}
```

図 7

```
void apply_stencil(dense<f64, 2>& grid,
i32 iterations) {
    _for(i32 i = 0, i < iterations, ++i) {
        map(stencil)(grid);
    } _end_for
}
void stencil(f64& cell) {
    arbb::array<usize, 2> coord;
    position(coord);
    usize x = coord[0], usize y = coord[1];
    _if(x != 0 && y != 0 && x != WIDTH-1
&& y != HEIGHT-1) {
        cell = 0.25 * (neighbor(cell, -1,
0) + neighbor(cell, 1, 0) +
 neighbor(cell, 0, -1) + neighbor(cell, 0,
1));
    } _end_if
}
```

図 9

```
void matvec_product(const dense<f32,
2>& matrix, const dense<f32>& vector,
dense<f32>& result)
{
    result = add_reduce(matrix * repeat_
row(vector, matrix.num_rows()));
}
```

図 8

「_for と arbb::map を組み合わせて使用すると、プログラマーは、タスクを管理することなく、ベクトル化とスレッド化を行うことができます。」

シリアル / 並列アプリケーションのパフォーマンス解析の簡素化

Levent Akyil

操作性が向上したインテル® VTune™ Amplifier XE の直感的なユーザー・インターフェイスにより、マウスを数回クリックするだけで強力なパフォーマンス解析を実行できます。

インテル® VTune™ Amplifier XE は、ソフトウェア開発者がアプリケーションのアルゴリズムやマイクロアーキテクチャーの問題を特定できるように支援する、強力なパフォーマンス解析ツールです。操作性が向上した**インテル® VTune™ Amplifier XE** の直感的なユーザー・インターフェイスにより、マウスを数回クリックするだけで強力なパフォーマンス解析を実行できます。アルゴリズムとマイクロアーキテクチャーのパフォーマンス解析のために、革新的で、実行が容易な解析タイプが用意されています。アルゴリズム・チューニング向けの定義済みの解析タイプは、ライト hotspot 解析、hotspot 解析、コンカレンシー解析、ロックと待機の解析の 4 つです。

- hotspot 解析は、開発者がアプリケーション・フロー（つまり、コールスタック情報）を理解し、オーバーヘッドの低い統計サンプリング（ユーザーモード・スタック・サンプリング）テクノロジーを利用して、実行に長い時間のかかるコードのセクション（hotspot）を特定できるよう支援します。
- コンカレンシー解析およびロックと待機の解析は、hotspot 関数とコールスタックの特定に加え、スレッド・プロファイリング・テクノロジーを利用して、指定されたシステムで利用可能なプロセッサーをアプリケーションがどのように使用しているか測定します。
  - コンカレンシー解析は、プロセッサー使用率が低い場所と、いつ、どのようにスレッドが実行、同期、待機しているかを特定します。
  - ロックと待機の解析は、プロセッサー使用率が効率的ではない原因を特定します。プロセッサーの使用率が低い最も一般的な原因は、スレッドが同期オブジェクト（ロック）を長時間待っているためです。
- ライト hotspot 解析は hotspot 解析と似ていますが、代わりにハードウェア・ベースのイベントベース・サンプリング (EBS) テクノロジーを使用して、指定されたアプリケーションの hotspot を特定します。図 1

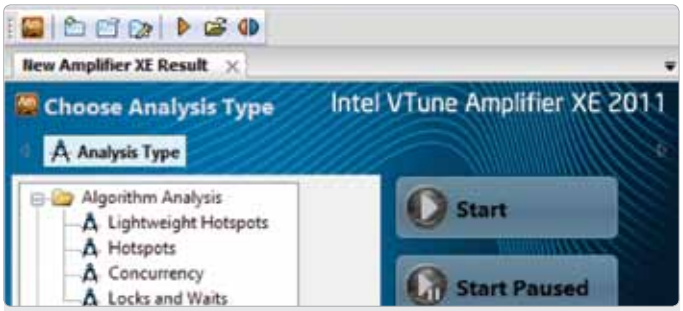


図 1: 定義済みのアルゴリズム解析 : ライト hotspot、hotspot、コンカレンシー、ロックと待機

より高度で詳細なマイクロアーキテクチャー解析のため、ツールには定義済みの解析タイプが用意されています。これは、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 : SandyBridge）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。これらの高度な定義済みの解析タイプが使用されると、ツールは問題の関数をハイライトしてヒントと提案を表示します。

この記事では、開発者が計算の問題を特定し、FLOPS（1 秒間の浮動小数点演算回数）のような主要なパフォーマンス評価基準を推定するのに、インテル® VTune™ Amplifier XE の高度な機能と EBS テクノロジーがどのように役立つかについて説明します。FLOPS（フロップス）は、ハイパフォーマンス・コンピューティングで頻繁に使用される性能指標で、指定されたマイクロプロセッサーのパフォーマンスと計算能力を測定する一般的な方法です。hotspot、コンカレンシー、ロックと待機などの別の解析タイプは、ユーザーモード・スタック・サンプリングと API インストルメンテーション・テクノロジーを使用します。

計算の問題の特定

パフォーマンス・チューニングは通常、十分に定義されたワークロードを完了するためにかかる時間を減らすことに焦点を置きます。パフォーマンス・イベントは経過時間の測定に使用できるため、ワークロードが完了するまでの経過時間を減らすことは、測定されるプロセッサー・サイクル（クロック数）を減らすことと同じです。インテル® VTune™ Amplifier XE の定義済み解析タイプであるライト hotspot は、プロセッサー・サイクルとリタイアした命令 [1] を使用してアプリケーションを解析します。サイクル数（クロック数）は、プログラムの実行にかかる時間を測定するための基本的な評価基準です。総サイクル数は、指定したアプリケーションの開始から終了までに費やされたサイクル数の合計です。一般的なパフォーマンス・チューニングでは、評価基準の総サイクル数をクロック数で測定できます。

マイクロアーキテクチャー解析と最適化の目標の 1 つは、マイクロオペレーションがディスパッチされなかったサイクルを特定することです。マイクロオペレーション（μ ops）は、より複雑な命令を実行するために使用される単純なマイクロプロセッサー命令です。

μ ops がディスパッチされなかったサイクルはストールサイクルと呼ばれ、表 1 のハードウェア PMU イベントを使用してカウントすることができます。これらのストールは、プロセッサーの実行ユニットのボトルネックの原因となり得ます。実行ユニットによって処理能力が決定されるため、本質的に実行ユニットは常にボトルネックであるといえ、アプリケーションの実行速度はそのボトルネックに左右されます。したがって、ストールサイクルの原因を特定して、可能であれば解決することが非常に重要です。要するに、実行ユニットはどんな理由であれ、アイドル状態や待機状態のままであってはなりません。

インテル® Core™2 プロセッサー・ファミリー（インテル® Core™2 Duo/Quad）	DIV	実行された除算の数をカウントします。実行された整数の除算、浮動小数点の除算、平方根の演算が含まれます。
	CYCLES_DIV_BUSY	除算または平方根の演算を実行するときに除算器がビジーだったサイクル数をカウントします。除算は、整数、X87 またはストリーミング SIMD 拡張命令 (SSE) のいずれかです。平方根の演算は、X87 または SSE のいずれかです。
インテル® Core™ アーキテクチャー（インテル® Core™ i7、i5、i3 プロセッサー（開発コード名 : Nehalem））	ARITH.DIV	実行された除算または平方根の演算の数をカウントします。除算は、整数、X87 または SSE のいずれかです。平方根の演算は、X87 または SSE のいずれかです。
	ARITH.CYCLES_DIV_BUSY	除算または平方根の演算を実行するときに除算器がビジーだったサイクル数をカウントします。除算は、整数、X87 または SSE のいずれかです。平方根の演算は、X87 または SSE のいずれかです。
第 2 世代インテル® Core™ アーキテクチャー（開発コード名 : SandyBridge）	ARITH.FP_DIV	実行された除算の数をカウントします。
	ARITH.FPU_DIV_ACTIVE	除算または平方根の演算を実行するときに除算器がビジーだったサイクル数をカウントします。

表 1: PMU イベントは特定のアーキテクチャーのイベントをカウントするために使用されます。

ストールサイクルと実行ユニットの使用率が最適でないことには多くの要因があります。例えば、メモリーアクセス（キャッシュミスなど）、分岐予測ミス（その結果パイプラインのフラッシュが発生）、計算の問題（除算、浮動小数点コントロール・ワードの変更などのレイテンシーが長い操作）、アウトオブオーダー(OOO) エンジンによりリタイアしない μ ops などがあります。除算や平方根などのレイテンシーが長い命令は、実行中にストールの原因になります。インテル® VTune™ Amplifier XE は、そのような操作がどこで行われているか、それらの操作が実行中のストールサイクルの原因になっているかどうかを正確に示します。

```
...
// 時間ステップの固定範囲でシミュレーションを実行
for (double s = 0.; s < STEPLIMIT; s += TIMESTEP)
{
  // ボディーの加速度を計算
  for (i = 0; i < n - 1; ++i)
  {
      for (j = i + 1; j < n; ++j)
      {
          // ボディー間の距離を計算
          double dx = body[i].pos[0]-body[j].pos[0];
          double dy = body[i].pos[1]-body[j].pos[1];
          double dz = body[i].pos[2]-body[j].pos[2];

          double distsq = dx*dx + dy*dy + dz*dz;
          if (distsq < MINDIST) distsq = MINDIST;
          double dist = sqrt(distsq);

          // j から i の単位ベクトルを計算
          double ud[3];
          ud[0] = dx / dist;
          ud[1] = dy / dist;
          ud[2] = dz / dist;

          // F = G*mi*mj/distsq だが F = ma なので ai = G*mj/distsq
          double Gdivd = GFORCE/distsq;
          double ai = Gdivd*body[j].mass;
          double aj = Gdivd*body[i].mass;

          // 単位ベクトルを使用して加速度コンポーネントを適用
          for (int k = 0; k < 3; ++k)
          {
              body[j].acc[k] += aj*ud[k];
              body[i].acc[k] -= ai*ud[k];
          }
      }
  }
  // 加速度を適用してボディーを進める
  for (i = 0; i < n; ++i)
  {
      for (j = 0; j < 3; ++j)
      {
          body[i].vel[j] += body[i].acc[j] * TIMESTEP;
          body[i].pos[j] += body[i].vel[j] * TIMESTEP;
          body[i].acc[j] = 0.;
      }
  }
}

...
```

図 2

例として、多体問題について考えてみましょう。多体問題は、互いの重力が相互に作用する天体の動きを予測します。サンプル・アプリケーションは時間ステップで進み、各ステップですべての天体の純粋な重力を計算して、位置、加速度、および速度を更新します。図 2 で示されているように、このシリアル実装では各反復で O(N2) 操作が必要です。

図 3 は、インテル® Core™ i7 プロセッサー (x980) ベースのシステム (3.33GHz、6 コア + ハイパースレッディング・テクノロジー有効) でインテル® VTune™ Amplifier XE を使用してサンプルコードを解析した結果です。

コードを最適化する 1 つの方法は、図 4 のように除算を逆数の乗算に置換することです。

最適化されたコードではクロック数が 46 億 6,800 万少なくなり、ディスパッチ・ストールが 74 億 4,800 万サイクルから 30 億 2,000 万サイクルに減少しています。図 5

さらに、アプリケーションを並列化することで、利用可能なコアを活用するだけでなく、DIV の影響を減らすことができます。並列化により、除算を実行するコアのすべてのポートが使用されます。

FLOPS の推定

このセクションでは、ハードウェア・ベースの EBS テクノロジーを使用してアプリケーションで実行される FLOPS を推定する方法を説明します。FLOPS は、32 ビットおよび 64 ビットの浮動小数点演算を指し、演算は加算または乗算のいずれかになります。

図 6、7、および 8 で示されているように、FLOPS はコンパイラーがコードを生成する方法に依存し、x87 レジスターまたは SSE レジスターで実行されます。浮動小数点命令が SSE レジスターで実行される場合、スカラーまたはパックド演算のいずれかになります。

表 2 と表 3 は、ハードウェアによって実行された計算上の浮動小数点演算を統計的に推定するために使用できる PMU イベント名を示しています。アーキテクチャーの投機的実行により、実行された命令がすべてリタイアされるとは限らないことに注意してください。したがって、これらのイベントのカウント数が実際の数よりも多くなることがあります。

インテル® VTune™ Amplifier XE は、これらのイベントを個別に、またはすべて同時に使用して、ハードウェアによって実行された FLOPS を推定します。経過時間の測定には、CPU_CLK_UNHALTED（クロック数）イベントを使用できます。プロセッサーの周波数が測定期間中に一定なのであれば、クロック数イベントを使用して経過ウォールクロック時間を計算できます。

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（英語）を参照してください。

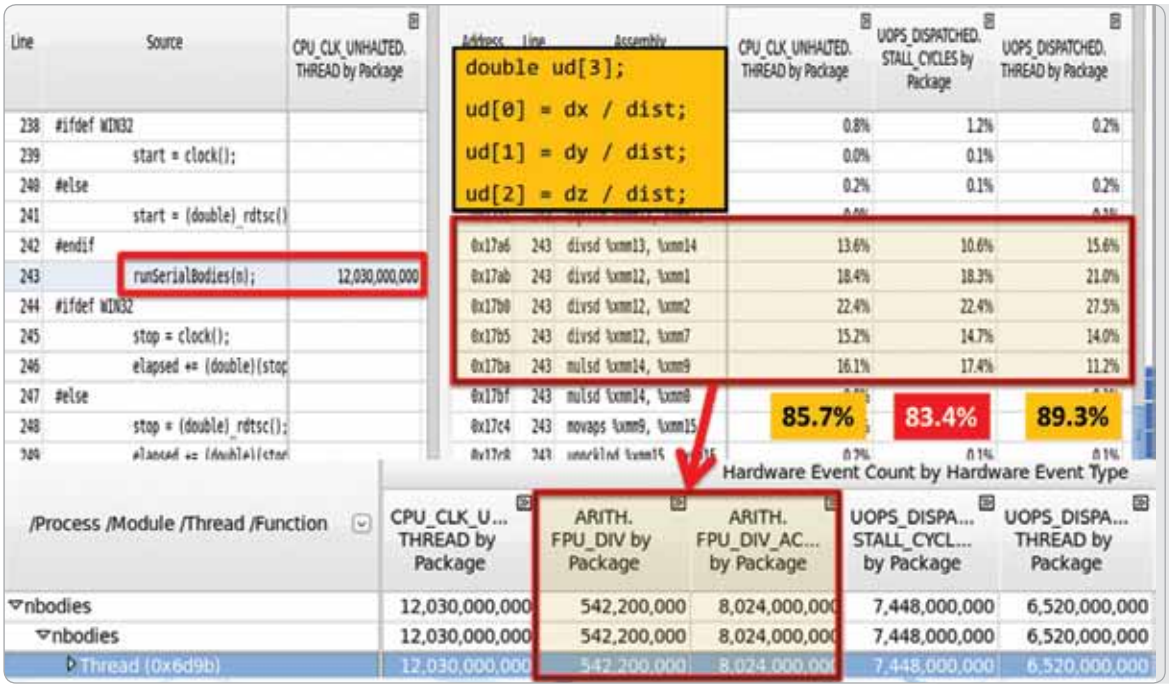


図 3:
サンプルコードの解析 — クロック数の 85.7%、総 μops ディスパッチの 89.3%、ディスパッチの 83.4% のストールがこのコードセグメントで発生しています。

```
// j から i の単位ベクトルを計算
double ud[3];
ud[0] = dx / dist;
ud[1] = dy / dist;
ud[2] = dz / dist;
```

```
// // j から i の単位ベクトルを計算
double ud[3];
double dd=1.0 / dist;
ud[0] = dx * dd;
ud[1] = dy * dd;
ud[2] = dz * dd;
```

図 4

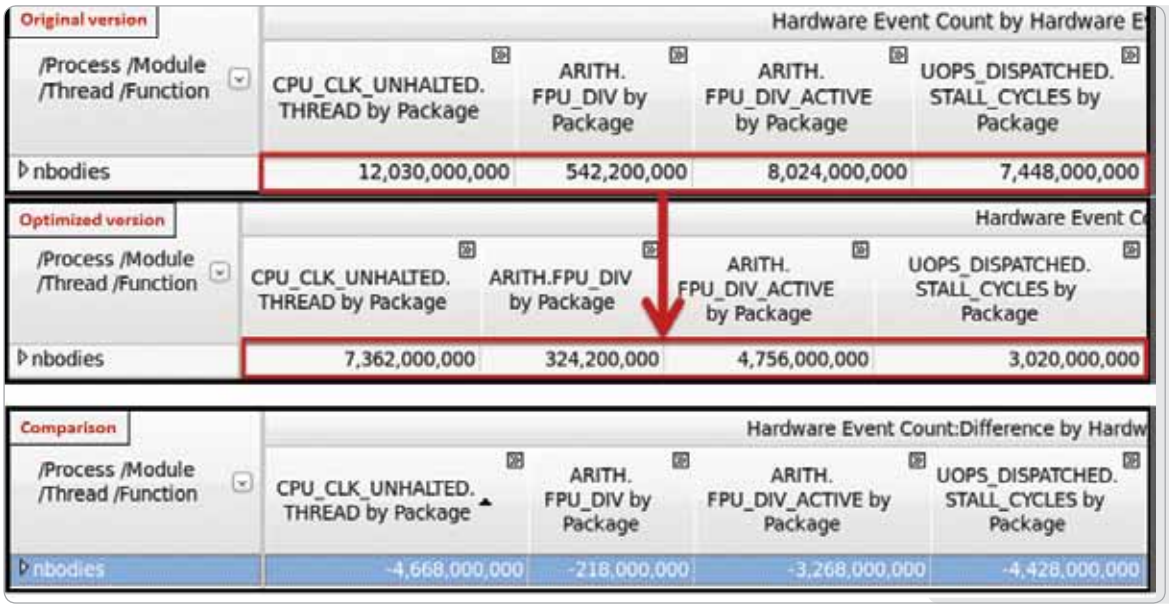


図 5:
オリジナルのバージョンと最適化されたバージョンの比較

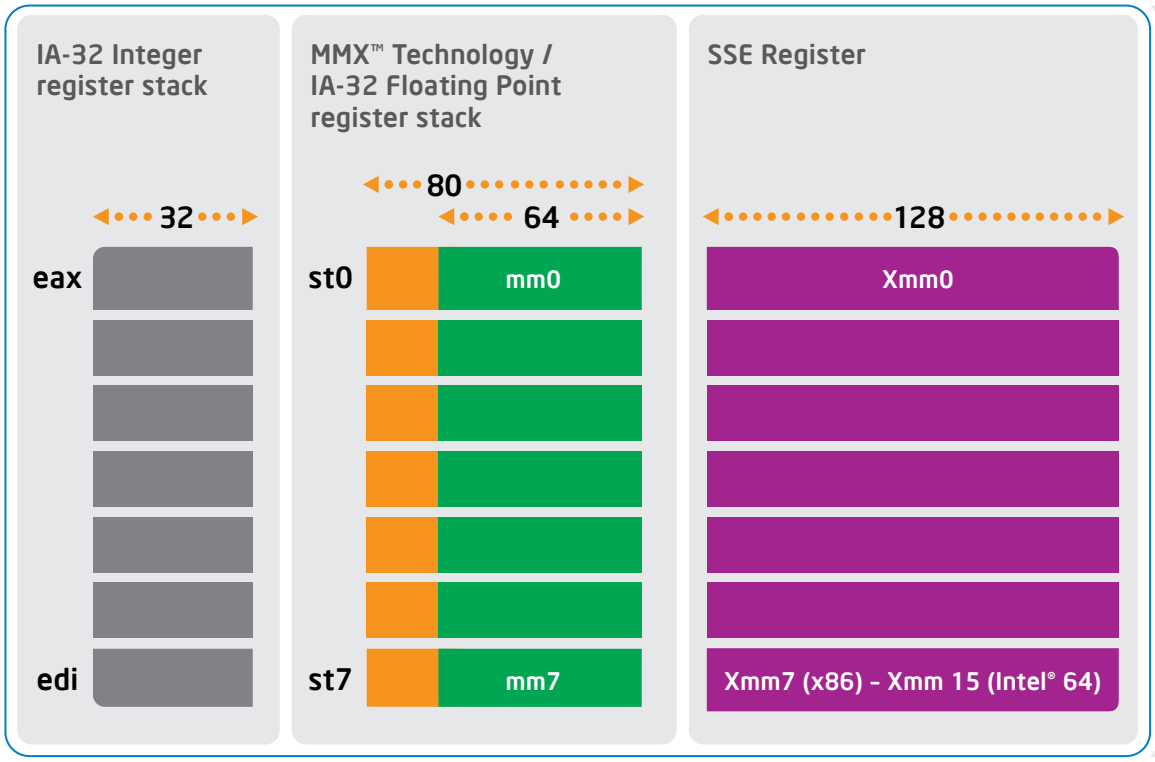


図 6:
インテル® アーキテクチャーの整数、浮動小数点、MMX、SSE、アドバンスト・ベクトル・エクステンション (AVX) レジスター。

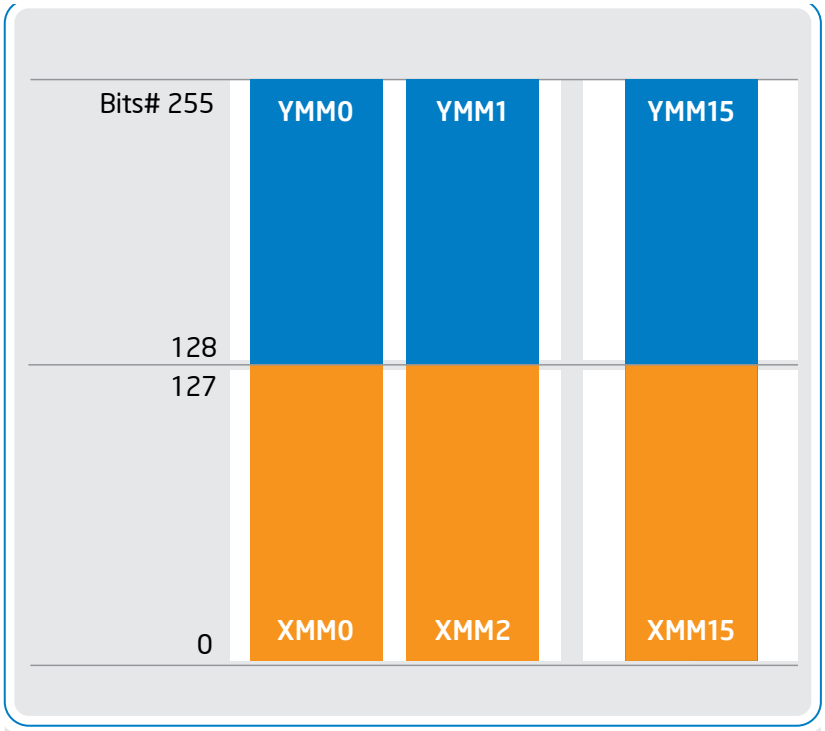


図 7:
インテル® AVX は、256 ビット幅の SIMD レジスターをサポートしています (32 ビット以下のモードでは YMM0-YMM7、64 ビット・モードでは YMM0-YMM15)。YMM レジスターの下位 128 ビットは 128 ビットの XMM レジスターにエイリアスされます。

「パフォーマンス・チューニングは通常、十分に定義されたワークロードを完了するためにかかる時間を減らすことに焦点を置きます。」

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 (英語) を参照してください。

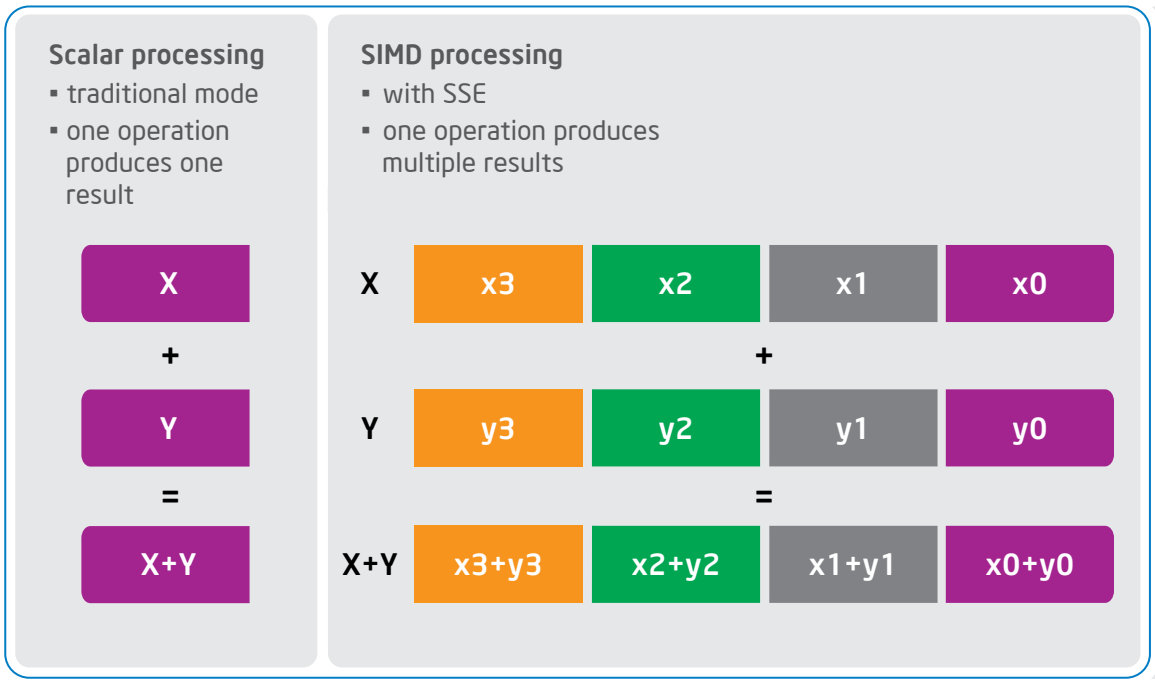


図 8: スカラー処理と SIMD (Single Instruction Multiple Data) 処理

プロセッサーの世代	プロセッサーのイベント名		
	従来の x87 を使用した浮動小数点演算	SIMD を使用した浮動小数点演算	
インテル® Core™2 プロセッサー・ファミリー (インテル® Core™ 2 Duo/Quad)	X87_OPS_RETIRED.ANY	パックド 64 ビット	SIMD_COMP_INST_RETIRED.PACKED_DOUBLE
		パックド 32 ビット	SIMD_COMP_INST_RETIRED.PACKED_SINGLE
		スカラー 64 ビット	SIMD_COMP_INST_RETIRED.SCALAR_DOUBLE
		スカラー 32 ビット	SIMD_COMP_INST_RETIRED.SCALAR_SINGLE
インテル® Core™ アーキテクチャー (インテル® Core™ i7、i5、i3 プロセッサー (開発コード名 : Nehalem))	FP_COMP_OPS_EXE.x87	パックド 64 ビット	FP_COMP_OPS_EXE.SSE_DOUBLE_PRECISION
		パックド 32 ビット	FP_COMP_OPS_EXE.SSE_SINGLE_PRECISION
		スカラー 64 ビット	FP_COMP_OPS_EXE.SSE_FP_SCALAR
		スカラー 32 ビット	FP_COMP_OPS_EXE.SSE_FP_SCALAR
第 2 世代 インテル® Core™ アーキテクチャー (開発コード名 : SandyBridge)	FP_COMP_OPS_EXE.X87	パックド 64 ビット	FP_COMP_OPS_EXE.SSE_PACKED_DOUBLE
		パックド 32 ビット	FP_COMP_OPS_EXE.SSE_PACKED_SINGLE
		スカラー 64 ビット	FP_COMP_OPS_EXE.SSE_SCALAR_DOUBLE
		スカラー 32 ビット	FP_COMP_OPS_EXE.SSE_SCALAR_SINGLE

表 2: PMU イベントは計算上の浮動小数点演算をカウントするために使用されます。

あるいは、参照サイクルの数をカウントしてスレッドの周波数の変更による影響を受けない、CPU_CLK_UNHALTED.REF を使用します。参照クロック数イベントとクロック数イベントの違いは、(HLT 命令を実行して) スレッドが停止状態になっても、参照クロック数イベントは、あたかもスレッドが最大周波数で連続して実行されているかのように、カウントを続行することです。

FLOPS の数式は以下のようになります。

FLOPS = ((FP 演算数 / クロック) * 計算上の FP 演算の総数) / 経過時間

経過時間 = CPU_CLK_UNHALTED / プロセッサーの周波数 / コア数。この数式では、ゼロでない CPU_CLK_UNHALTED イベントカウントのコアを考慮する必要があります。

ここでは、EBS テクノロジーによって FLOPS を推定する方法を説明するために、単純なマルチスレッドの行列乗算を使用します。このサンプル・アプリケーションは、スレッドプールの概念を使用して、スレッドプールの各スレッドで図 9 のコードを実行します。

アプリケーションは、浮動小数点演算の総数 (2 / 反復 * NUM * NUM * NUM) を経過時間で割って測定した FLOPS もレポートします。経過時間は行列乗算部分のみを含み、初期化やスレッド生成のオーバーヘッドは含みません。

適切なコードセクションのサンプルを収集するため、__itt_pause() (収集を休止) および __itt_resume() (収集を再開) API を使用しています。ユーザー API の使用方法については、インテル® VTune™ Amplifier XE のドキュメントを参照してください。

インテル® Xeon® プロセッサー上のインテル® VTune™ Amplifier XE では、図 10 のように設定します。

x87 レジスターの使用

サンプル・アプリケーションは、Windows* システムで Microsof* Visual Studio* を使用してリリースモードで (最適化レベルを 0x に設定して) コンパイルしました。

図 11 と 図 12 は、インテル® VTune™ Amplifier XE で解析したときのアプリケーションの様子を示しています。コンパイラーがコードを生成した方法についての情報が提供されています。この実行では、x87 レジスタースタックを使用して FLOPS のサンプルを収集したことが分かります。

次に、図 13 の数式に数字を設定します。

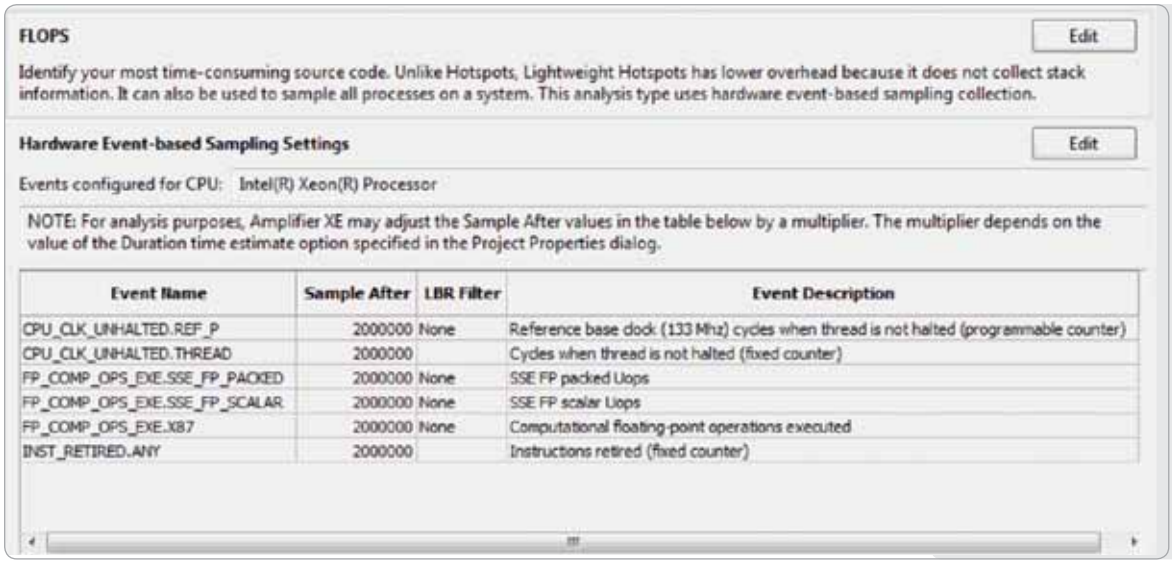
プロセッサーの世代	プロセッサーのイベント名	
	AVX を使用した浮動小数点演算	
第 2 世代インテル® Core™ アーキテクチャー (開発コード名 : SandyBridge)	パックド 64 ビット	SIMD_FP_256.PACKED_DOUBLE
	パックド 32 ビット	SIMD_FP_256.PACKED_SINGLE
	注 : パックド AVX-256 は 1 としてカウントでき、SIMD FP 128 をカウントします。	

表 3: PMU イベントは、AVX を使用した計算上の浮動小数点演算をカウントするために使用されます。

```
double a[NUM][NUM];
double b[NUM][NUM];
double c[NUM][NUM];
 ...
slice = (unsigned int) tid;
from  = (slice * NUM) / NUM_THREADS;
to    = ((slice + 1) * NUM) / NUM_THREADS;
for(i = from; i < to; i++)
{
      for(j = 0; j< NUM; j++)
      {
          for(k = 0; k < NUM; k++)
              // 2 浮動小数点演算 / 反復:1 加算、1 乗算
              c[i][j] += a[i][k] * b[k][j];
      }
}
 ...
```

図 9

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 (英語) を参照してください。



FLOPS Edit

Identify your most time-consuming source code. Unlike Hotspots, Lightweight Hotspots has lower overhead because it does not collect stack information. It can also be used to sample all processes on a system. This analysis type uses hardware event-based sampling collection.

Hardware Event-based Sampling Settings Edit

Events configured for CPU: Intel(R) Xeon(R) Processor

NOTE: For analysis purposes, Amplifier XE may adjust the Sample After values in the table below by a multiplier. The multiplier depends on the value of the Duration time estimate option specified in the Project Properties dialog.

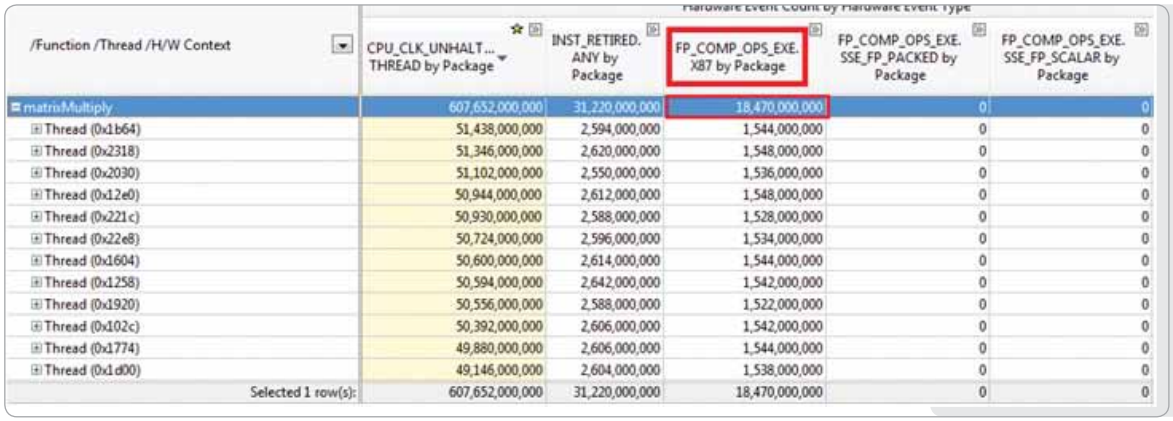
Event Name	Sample After	LBR Filter	Event Description
CPU_CLK_UNHALTED.REF_P	2000000	None	Reference base clock (133 Mhz) cycles when thread is not halted (programmable counter)
CPU_CLK_UNHALTED.THREAD	2000000		Cycles when thread is not halted (fixed counter)
FP_COMP_OPS_EXE.SSE_FP_PACKED	2000000	None	SSE FP packed Uops
FP_COMP_OPS_EXE.SSE_FP_SCALAR	2000000	None	SSE FP scalar Uops
FP_COMP_OPS_EXE.X87	2000000	None	Computational floating-point operations executed
INST_RETIRED.ANY	2000000		Instructions retired (fixed counter)

図 10: 適切な PMU イベントを測定するために作成されたカスタム解析タイプ

```
D:\Examples\flops\msvc\Release>flops.exe
MFLOPS:1130.180 mflops
Thread #:12 Elapsed time = 15.201000 seconds

D:\Examples\flops\msvc\Release>_
```

図 11



/Function /Thread /H/W Context	CPU_CLK_UNHALT... THREAD by Package	INST_RETIRED. ANY by Package	FP_COMP_OPS_EXE. X87 by Package	FP_COMP_OPS_EXE. SSE_FP_PACKED by Package	FP_COMP_OPS_EXE. SSE_FP_SCALAR by Package
matrixMultiply	607,652,000,000	31,220,000,000	18,470,000,000	0	0
Thread (0x1b64)	51,438,000,000	2,594,000,000	1,544,000,000	0	0
Thread (0x2318)	51,346,000,000	2,620,000,000	1,548,000,000	0	0
Thread (0x2030)	51,102,000,000	2,550,000,000	1,536,000,000	0	0
Thread (0x12e0)	50,944,000,000	2,612,000,000	1,548,000,000	0	0
Thread (0x221c)	50,930,000,000	2,588,000,000	1,528,000,000	0	0
Thread (0x22e8)	50,724,000,000	2,596,000,000	1,534,000,000	0	0
Thread (0x1604)	50,600,000,000	2,614,000,000	1,544,000,000	0	0
Thread (0x1258)	50,594,000,000	2,642,000,000	1,542,000,000	0	0
Thread (0x1920)	50,556,000,000	2,588,000,000	1,522,000,000	0	0
Thread (0x102c)	50,392,000,000	2,606,000,000	1,542,000,000	0	0
Thread (0x1774)	49,880,000,000	2,606,000,000	1,544,000,000	0	0
Thread (0x1d00)	49,146,000,000	2,604,000,000	1,538,000,000	0	0
Selected 1 row(s):	607,652,000,000	31,220,000,000	18,470,000,000	0	0

図 12: インテル® VTune™ Amplifier XE で x87 レジスタースタックを使用するアプリケーションを解析した結果を示しています。matrixMultiply() 関数がすべてのスレッドでほぼ等しく実行されています。

MFLOPS 数式 = FP_COMP_OPS_EXE.FP / 1x10⁶ / 経過時間
経過時間 = CPU_CLK_UNHALTED.THREAD /
プロセッサーの周波数 / コア数

経過時間 = 607,652,000,000.00 / 3.33 x 10⁹ / 12 = 15.206 秒
MFLOP = 18,470,000,000.00 / 1x10⁶/ 15.206 秒 = 1214.652 Mflops

図 13

```
D:\Examples\flops\msvc\Release>flops.exe
MFLOPS:10437.345 mflops
Thread #:12 Elapsed time = 1.646000 seconds

D:\Examples\flops\msvc\Release>_
```

図 14

/Function /Thread /H/W Context	Hardware Event Count by Hardware Event Type				
	CPU_CLK_UNHALTED. THREAD by Package	INST_RETIRED. ANY by Package	FP_COMP_OPS_EXE. X87 by Package	FP_COMP_OPS_EXE. SSE_FP_PACKED by Package	FP_COMP_OPS_EXE. SSE_FP_SCALAR by Package
threadPool	66,178,000,000	20,460,000,000	0	9,152,000,000	0
Thread (0x1f90)	5,810,000,000	1,714,000,000	0	774,000,000	0
Thread (0x1b0c)	5,774,000,000	1,702,000,000	0	768,000,000	0
Thread (0x14f8)	5,724,000,000	1,698,000,000	0	762,000,000	0
Thread (0x1e64)	5,708,000,000	1,706,000,000	0	774,000,000	0
Thread (0x1a3c)	5,652,000,000	1,704,000,000	0	768,000,000	0
Thread (0x2348)	5,600,000,000	1,720,000,000	0	760,000,000	0
Thread (0x1048)	5,516,000,000	1,700,000,000	0	756,000,000	0
Thread (0x1430)	5,502,000,000	1,702,000,000	0	762,000,000	0
Thread (0x188c)	5,482,000,000	1,714,000,000	0	776,000,000	0
Thread (0x8f0)	5,442,000,000	1,710,000,000	0	746,000,000	0
Thread (0xc54)	5,000,000,000	1,700,000,000	0	752,000,000	0
Thread (0x21b0)	4,968,000,000	1,690,000,000	0	754,000,000	0

図 15:
インテル® VTune™ Amplifier XE で SSE レジスターを使用するアプリケーションを解析した結果

図 16:
関数呼び出しと関連付けられているイベント

MFLOPS 数式 = 2 * FP_COMP_OPS_EXE.SSE_DOUBLE_PRECISION / 経過時間
経過時間 = CPU_CLK_UNHALTED.THREAD / プロセッサーの周波数 / コア数
経過時間 = (66,178,000,000 / 3.33 x10^9 / 12) = 1.656 秒
MFLOPS = 2 * FP_COMP_OPS_EXE.SSE_DOUBLE_PRECISION / 1 x 10^6 / 1.656 秒 = 11053.140 mflops

図 17

SSE レジスターの使用

次に、同じアプリケーションで SSE レジスターを使用した場合を見てみましょう。インテル ® コンパイラー 12.0 を使用してコンパイルしたアプリケーションをインテル® VTune™ Amplifier XE で実行すると、結果は図 14 のようになります。

図 15 の新しい結果を見ると、サンプルが表示されている関数名が違っていることが分かります。以前の例（図 12 を参照）では matrixMultiply 関数のサンプルが表示されていましたが、今回は threadPool 関数のサンプルが表示されています。これはインライン展開によるものです。threadPool の関数呼び出しをドリルダウンするとよく分かります。

さらに、このバージョンでは、各スレッドで費やされたクロック数が（x87 レジスタースタックを使用したバージョンの）500 億サイクルから 50 億サイクルに減少していることが分かります（図 15 を参照）。この向上はベクトル化や SSE レジスターの使用によるものだけではないことに注意してください。インテル® コンパイラーによって行われたほかの最適化も、この向上に役立っています。

図 17 では、2 つのパックド倍精度浮動小数点演算を 128 ビット XMM レジスターで実行できるため、FP_COMP_OPS_EXE.SSE_DOUBLE_PRECISION イベントに 2 を掛けています。単精度浮動小数点演算の場合、パックド単精度浮動小数点の総数に 4 を掛ける必要があります。

この記事では、例を使って、インテル® VTune™ Amplifier XE の強力な EBS テクノロジーにより長いレイテンシーの命令が原因の計算問題を特定する方法と、FLOPS を推定する方法を説明しました。ここでは、インテル® VTune™ Amplifier XE の高度な機能をいくつか示しましたが、インテル® VTune™ Amplifier XE の直感的で使いやすい定義済みの解析タイプを使用すれば、シリアル・アプリケーションと並列アプリケーションのパフォーマンス解析を簡単に行うことができます。並列アプリケーションは、並列化の複雑さのために独自の問題が発生しがちです。インテル® VTune™ Amplifier XE は、これらの問題に取り組むために設計されています。問題の解決に不可欠なプロファイリング・ツールとさまざまなテクニックを提供することで、開発者がシリアル・アプリケーションと並列アプリケーションのボトルネックを特定できるように支援します。

詳細は、製品のサイトを参照してください。o

1. リタイアした命令：最新のインテル ® 64 および IA-32 プロセッサーは、アウトオフオーダー実行エンジンを使用したマイクロアーキテクチャーを採用しています。さらに、プログラムを順に実行するインオーダーのフロントエンドとリタイアメント・ロジックも含まれます。実行が完了した命令は、リタイアした命令と呼ばれます。

「並列化により、除算を実行するコアのすべてのポートが使用されます。」

DEVELOPER SPOTLIGHT

池井 満
ソフトウェア & サービス
エンジニアリング・マネージャー

現在の取り組み

大学や研究機関の計算機センターで使われる計算機の性能評価をお手伝いしています。いくつかの機関の評価用のプログラムを AVX に最適化して、要望通りの性能を達成することができました。結果、SandyBridge† を用いた次世代のサーバーでもさらなる性能向上が得られることを示すことができました。

最近学んだ開発者にとって興味深いこと

インテル® Xeon プロセッサーのクラスターやメニーコアを応用する HPC の分野と、携帯やタブレットといった小型端末に関する分野の2つのプロジェクトを担当しています。一見まったく違うように見える分野ですが、実は両者とも消費電力当りの性能が最も重要で、プロセッサーのアーキテクチャー上は似たような部分があることが分かり、面白いと思いました。

エンジニアリング・マネージャーとして最大の課題とその対処法

プロジェクトの成果を得るために、グループ外へのビジビリティーを大きくすることとグループ内の技術者を活性化することです。達成した仕事を日本の営業部隊に分かりやすく伝えることは重要です。また、さまざまな状況において、技術者が興味を保ちながら仕事をできるよう注意しています。トレーニングは特に重要です。

業界を揺るがす重要なトレンドと現在の業務へのその影響

最近、低消費電力製品のインテル® Atom™ プロセッサー・ファミリーが使われるようになってきました。これまでは、大容量のメモリーや高速のネットワーク接続の環境を前提に、PC やサーバーの構成でソフトウェアの最適化を行ってきました。これからは、電池の容量や、センサーの応答、低速のメモリーといった特定のハードウェア構成の制限上で最適化を行う必要があります。

† 開発コード名

ケーススタディー：

マサチューセッツ総合病院

Bevin Brett

マサチューセッツ総合病院とインテルの開発者が仮想大腸内視鏡検査の処理を向上させた事例を紹介します。

マサチューセッツ総合病院 (MGH) の 3D Imaging Research 研究チーム（Dr. Hiro Yoshida PhD.、Dr. Yin Wu PhD. ほか）は、仮想大腸内視鏡検査を行うハードウェアとプロトタイプのソフトウェアを所有していました。しかし、その実行と表示には 45 分以上もかかっていたため、実際に現場で使用するには遅すぎました。

Steve Aylward 氏 (Microsoft* 社 General Manager for Commercial Health and Life Sciences) は、eWeek.com の記事で次のように述べています。

> *マサチューセッツ総合病院では、侵襲的処置、下剤、鎮静剤の使用、また従来の大腸内視鏡検査の高い費用を避ける一方、仮想大腸内視鏡検査の処理を向上させる方法について Microsoft とインテルにアドバイスを求めていました。*

MGH とインテルの開発者は、最新のインテル® Parallel Studio 製品を使用してシリアル・アルゴリズムを解析し、C++ コードを変更して、並列化を行いました。チームは、会議でアプリケーションのデモを 5 分未満でできるように、実行と表示時間を短縮することを目標に作業を開始しました。変更は数ヶ月かけて行われ、厳しい期限に遅れることなく、当初の目標を達成することができました。変更後のコードは、SC 2010* と RSNA 2010* のデモンストレーションで使用されました。

この記事では、C++ プログラムのパフォーマンスを大幅に向上させるために開発者が取り組んだ事例を紹介します。

注意

実際のプログラミングは、（特に複数の会社のチームで行う場合は）複雑です。時間の制約、学習曲線、ほかの仕事との調整など、理想的な順序で作業を行えないさまざまな要因があります。以下の説明は、すべての作業や作業が行われた正確な順序を再現しているものではないことに注意してください。

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（英語）を参照してください。

アプリケーション

仮想大腸内視鏡検査はすでに医療の現場で活用されていますが、患者が検査前日から絶食して下剤などを使用する必要があることは従来の検査と変わりません。改良されたプロトタイプの仮想大腸内視鏡検査では、患者は低繊維質の食品を食べて X 線を通さない着色剤を含む錠剤を飲むようになります。CT スキャンで患者の下腹部横断面の X 線画像を撮影した後、コンピューター・プログラムで画像を処理して着色された食品を消去します。医師は、仮想の大腸壁を見ながら、処理されたデータをインタラクティブに確認することができます。

仮想大腸内視鏡検査には 3 つのステップがあります。

1. CT スキャンのデータを、患者の腹部を含む立方体内の場所の密度を表す立方体データ構造に適用します。
2. 「電子洗浄」コードで立方体を処理して、画質処理された立方体を生成します。このステップは、大腸にある着色された食品を消去して一部の画像を拡張します。
3. 医師は、最終的な立方体の大腸の様子を見て、ポリープなどの気になる症状がないか調べます。

2 つ目のステップが最も重要な計算ステップです。計算はほかのステップでも行われますが、この記事ではこのステップに焦点を当てています。2 つ目のステップで、データは一連のステージを流れます。各ステージは前のステージで生成された 1 つ以上の立方体を受け取り、後のステージで使用する新しい立方体を生成します。

最初のステージの入力は、**DICOM** データ構造としてエンコードされる、患者の腹部内を均一に拡張したサンプルの密度データです。このデータは、患者の腹部のグレイスケール X 線画像（解像度 512x512）で、約 500 枚あります。各ボクセル（3 次元体積のピクセル）の値は -2000（空気密度）から 2000（X 線をほぼ完全に遮断）になります。骨と X 線の造影剤は通常、中間値になります。筋肉、水、脂肪の密度は骨未満で空気以上です。各ボクセルに記録される密度は、密度に関係するさまざまな物質の相対量に依存します。この形式では、未処理のデータは約 0.25GB（0.5K x 0.5K x 0.5K x 2 バイト）あります。

大量のデータはステージで費やされるプロセッサー時間に大きな影響を与えます。データ構造とアルゴリズムは、ディスク間で大量のデータを転送する時間とシステムのメモリー量を考慮する必要があります。

データの中間立方体の多くは、ボクセルを 4 バイトの float 型で表現します。このため、サイズは 1GB 近くになります。ただし、非ゼロの値になるのはそのうちの約 1/8 だけです。非ゼロの値をまとめて、ゼロのみの領域を省略するだけで、サイズは約 0.1 ～ 0.25GB に圧縮できます。

ステージを通して処理される立方体の数は 20 以上になり、さらに各ステージでいくつかの中間立方体が必要になります。その結果、データ量は 2GB から 5GB になり、複数の中間立方体を並列処理した場合は 10GB から 20GB まで増加します。

パイプを流れるデータはディスクで書き込み / 読み取り可能なため、パイプのステージはスタンドアロンで実行できます。最終データはディスクに書き込まれます。

最初に下した決定の 1 つは、64 ビットのアドレス空間を使用して、立方体をできるだけメモリーに保持することでした。データ構造にはポインターが比較的少なかったため、ポインターを長くしてもデータサイズはそれほど増加しませんでした。

立方体のデータ構造

立方体は C++ テンプレート・クラスとして実装され、テンプレート引数にはボクセルの種類を指定します。立方体内の任意の (X, Y, Z) インデックスの読み書きや、立方体を反復する操作は、さまざまなメンバー関数によって提供されます。

立方体はより小さな立方体へのポインターの配列として実装され、より小さな立方体にはボクセルの配列へのポインターが含まれます。これらのより小さな立方体も C++ テンプレート・クラスとして実装されます。スペースを節約するため、部分立方体を省略し、ボクセルの配列を共有します。

ZLIB の使用

アプリケーションを 64 ビットにするという決定に伴い、ハードウェア向けにすでに最適化されている 64 ビット・バージョンの **ZLIB** ライブラリーが必要になりました。この要件を完全に満たす最適化されたライブラリーは、**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）の サンプル**にありました。

コーディングの変更

64 ビットへの変更、ZLIB の導入、立方体テンプレート・クラスのさまざまな小さな改良に伴う問題の発生により、新しいコードをコンパイルし、実行できるようになるまで数日かかりました。

パイプラインにデータを流し、すべてのステージでデバッグを行い、特に問題が感じられなくなるまで、数週間かかりました。まず最初に感じたのは、初期ステージが速くなったことでした。

しかし、プロダクション・データセットでリリースビルドを実行すると、各ステージで 5 分以上かかりました。デバッグビルドを使用すると、時間はさらに遅くなり、一部のコード領域ではリリースビルドの 100 倍以上かかりました。このため、ファイルの多くは最適化してコンパイルし、デバッグする必要のあるファイルにのみデバッグビルドを使用しました。

前のステージのパフォーマンスを改善するとともに、後のステージのデバッグも並行して行いました。

チームは、パフォーマンス向上のために以下の手法を採用しました。

1. インテル® Parallel Advisor でアプリケーションをプロファイルして、トップダウンのコールツリーとループ表示を確認します。
2. まだ改善されていないが、シリアル時間の改善または並列化の実装のいずれかが見込める次のホットな外側のループを探します。並列化可能なアプローチを選択して、シリアル部分の改良を最初に行います。並列化を実装する場合は、サイト / タスク注釈を追加してサイトが適切であることを確認します。
3. そのループまたはループの周囲の単体テストを作成します。
4. 単体テストを測定します。
5. ループを変更します。
   a. 並列化を実装する場合は、インテル® Parallel Advisor を使用して起こり得る問題を調べます。
   b. シリアル部分を変更するか、並列化を実装します。
6. 並列化を実装した場合は、インテル® Parallel Inspector を使用して問題がないか調べます。インテル® Parallel Inspector は、データ競合の検出もサポートしています。
7. 単体テストがパスすることを確認します。
8. 単体テストを測定して、必要なスピードアップが達成されたかどうかを判断します。達成されなかった場合は、処理を繰り返します。

測定

（2 つ目の）電子洗浄ステップの目標は、5 分を切ることでした。

最初の測定結果は、非常に残念なものでした。パイプラインの 5 つのステージすべてで時間がかかっていました。ステージ内には時間のかかるループが合計で約 20 ありますが、ほとんどのループで 5 分以上かかっていたのです。大幅に経過時間を短縮するには、そのほとんどを改善する必要がありました。

単体テスト

大量の入力データのため、各ループは複雑なアルゴリズムを適用していました。入力データはパイプラインの初期のもので容易に改変できず、同じ特性を持つ適切な小さな入力セットを作成する方法は全く明らかではありませんでした。このため、単体テストを本物のデータで行う必要がありました。

変更するアルゴリズムごとに、単体テストが作成されました。コードは、アルゴリズムの入力をファイルに書き出すように変更され、単体テストは以下のように作成されました。

1. ディスクから入力を読み取ります。
2. 既存のアルゴリズムを入力に適用して、現在の「新しい」出力を取得します。
3. 既存のアルゴリズムのコピーを入力に適用して、古い出力を取得します。
4. 2 つの出力を比較して、出力が同じであることを確認します。

もちろん、出力は同じはずです。この比較は単体テストを実際にテストしており、すべての入力をキャプチャーできなかったケースを特定することができました。

その後、既存のアルゴリズムを新しいアルゴリズムに変更して単体テストを再実行し、同じ出力が生成されたことを確認しました。

インテル® IPP の使用

Partners HealthCare の研究者は、インテル® IPP の畳み込みが一連の処理で使用されているいくつかのアルゴリズムに適していることを認識していました。

畳み込みは 2 つの配列間の数学的な演算です。基本的に、1 つ目の配列の最後に、2 つ目の配列の要素と同じ長さになるようにゼロが追加されます。その後、2 つ目の配列はこの拡張配列に沿ってスライドされます。スライドの各位置で内積が計算され（対応する要素が乗算され、積が合計されます）、合計が出力の同じ位置に設定されます。

しかし、アルゴリズムは 1 次元の畳み込みではなく、次のようにして 3 次元の畳み込みを行っていました。まず最初に X 軸と平行な Y*Z 列に沿って移動し、中間結果を生成します。次に、この中間立方体の Y 軸と平行な X*Z 列に沿って移動し、別の中間結果を生成します。最後に、Z 軸と平行な X*Y 列に沿って移動し、最終的な結果を生成します。

その結果、3 つの次元の 3 つのループで、その時間のほとんどがインテル® IPP の畳み込みコード用に Float32s のショートベクトルを生成して列にインデックスを付ける処理に費やされていました。その後、値を元に戻すため、出力の立方体にインデックスを付けていました。

このアルゴリズムに対して 4 つの大幅な変更が行われました。

1. Float32s として中間結果を維持する。
2. インデックスを変更する。その結果、元の立方体のインデックスで見ると、最初に (X, Y, Z) として開始したセルは、最初の中間結果で (Y, Z, X) に書き込まれ、次の中間結果で (Z, X, Y) に書き込まれ、最後に最終結果で (X, Y, Z) に戻されます。
3. 畳み込み関数に使用するより長いデータのベクトルを組み立てる。

   以下のデータから成る 2 つの元になる列を考えてみます。

   (0,0,0,0, 0,**1**,0,0, 0,0,**2**,0, 0,0,0,0)

   (0,0,0,0, **1**,0,**1**,0, 0,**2**,**2**,0, 0,0,0,0)

   ぶれのために両方の列を長さ =3 のベクトル (0.25, 0.5, 0.25) で畳み込みます。

   2 つの列の間にゼロをパディングして、1 つの長いベクトルにします。生成されるベクトルを畳み込みコードで使用します。その後、パディングに対応する結果の部分は破棄されます。

   さらに、ゼロの続く部分を省略して、結果でこれらの省略された部分にゼロを使用します。

   最終バージョンでは、上記の 2 つのベクトルを畳み込みの入力にします。

   (0,0,**1**,0, 0,**2**,0, 0,**1**,0,**1**,0, 0,**2**,**2**,0, 0)

   畳み込みの結果は以下のように適用されます。

   (0,0.25,**0.5**,0.25,0.5,**1.0**,0.5,0.25,**0.5**,0.5,**0.5**,0.25,0.5,**1.5**,**1.5**,0.5,0.0)

   ゼロと列の最後が再度挿入され、最終的な答えが作成されます。

   (0,0,0,0, 0.25,**0.5**,0.25,0.0, …) および

   (…)
4. 立方体全体が一度に実行された場合、中間結果は 1 次または 2 次キャッシュに入らず、メインメモリーはほとんどのトラフィックでアクセスされます。代わりに、大きな立方体は小さな立方体に分解され、個別の問題として扱われます。この試みはより小さな立方体の分類に使用されました。立方体が小さすぎる場合、より小さな立方体の最後を超えて移動する処理が必要になるため、ハードウェア固有の最適なサイズがあります。

マルチコアでの並列化の利点

インテル® IPP の使用や、その他シリアル処理部分の拡張による変更によって、電子洗浄ステップで費やされていた時間は大幅に短縮されましたが、目標の 5 分を切るにはさらなる高速化が必要でした。

インテル® Parallel Advisor の調査機能により、並列化できる多くのループが指摘され、これらのループは一つずつ攻略されていきました。2 つのステージでは複数の個別の畳み込みを行い、ほかのステージでは立方体内のボクセルの大きなサブセットを反復するアルゴリズムを使用していました。

畳み込みの並列化

あるステージで 3 つの畳み込みが行われ、別のステージで 9 つの畳み込みが行われていました。どちらの場合も、ステージで行われた畳み込みはすべて、同じデータに対して行われていました。

並列化の導入には 3 つの選択肢がありました。

1. 個別の畳み込みを並列に行う。
2. 立方体の部分の 3D 畳み込みを並列に行う。
3. ベクトルの 1D 畳み込みを並列で行う。

1D 畳み込みはインテル® IPP ですでに行われているため、効率的にハードウェアを利用していると仮定しました。

9 つの個別の畳み込みを別のコアで実行すると、作業量が最小化され、最大で 9 倍高速化される可能性があることは明白でした。チームは、インテル® Parallel Advisor の正当性機能を使用して、個別に行われていた畳み込みを並列に実行し、どの程度高速化できるかを測定しました。

立方体を最適なサイズの立方体に分解した場合でも、16 コアのシステムで 9 つの畳み込みをすべて並列に実行すると、約 3 倍高速化されることが分かりました。読み書きするデータは非常に多く、データはほとんど再利用されていませんでした。メインメモリーの

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（英語）を参照してください。

帯域幅が複数の独立したタスクの処理を遅らせている要因であり、キャッシュはあまり役に立っていませんでした。

チームは、並列に実行される畳み込みの数と達成される高速化の関係についてさらに実験を行い、立方体を分解するサイズを注意深く選択したところ、このアルゴリズムとハードウェアでは、3 ウェイの並列化が最も適していることを発見しました。9 つの畳み込みは、それぞれ 3 つの畳み込みを行う 3 つのタスクに分散され、同様に、ほかのステージの 3 つの畳み込みも、それぞれ 1 つの畳み込みを行う 3 つのタスクに分散されました。

For-all-points アルゴリズムと wave-front アルゴリズム

立方体内の多くのボクセルを、セットにおけるボクセルの位置または状態に基づいて処理するアルゴリズムはいくつかあります。立方体のさまざまな部分を並列に処理することで、これらの処理をすべて行うことができます。

最初にロックが必要であると思われたケースでは、反復空間を連続したスライスに分割して、これらのスライスをロックの必要がない方法で処理することが可能でした。

具体的には、図 1 に示すように、外側の 3 つの反復ループ内でこれらのスライスを処理しました。

その後、inp と out の x-1、x、および x+1 要素がこのループの反復で共有されていないこと、つまりロックする必要がないことがわかったため、真ん中のループを並列化することができました。

結果

変更を重ねた結果、すべての並列化の候補を実際に並列化する前に、プログラムの実行時間が目標の 5 分を切るようになりました。その結果、チームは計画通りにアプリケーションのデモを行うことができました。

```
for (int interleave = 0; interleave <
3; interleave++)
      for ( int slice = interleave;
slice < xMax/sliceSize;
slice += 3)

for ( int x = (slice+0)*sliceSize;
x < min((slice+1)*sliceSize, xMax);
x++)

operateOn(
inp[x-1],inp[x],inp[x+1],
out[x-1],out[x],out[x+1]);
```

図 1

まとめ

インテル® Parallel Advisor を使用することで、改善を試みても成果が得られないような個所を避け、非決定性がもたらすクラッシュや誤った解が引き起こされる前に正当性を検証して、焦点を当てるべき個所を特定することができるようになりました。

インテル® Parallel Advisor をインテル® Parallel Inspector やインテル® Parallel Amplifier と組み合わせて使用することで、スケジュール通りに複雑なコードのパフォーマンスを大幅に向上させることができました。

アプリケーションのニーズを満たすインテル® IPP の 2 つの最適化機能（畳み込みと ZLIB）を使用することにより、チームは信頼性のある最適化されたコードを生成するための時間を大幅に節約することができました。

DEVELOPER SPOTLIGHT（続き）

池井 満
ソフトウェア & サービス
エンジニアリング・マネージャー

仕事におけるエキサイティングな点

まだ、世の中に出ていない最高性能のプロセッサーを使用して仕事ができることです。

今日の開発における並列化の重要性と将来のその役割

HPC 分野では、数年前まで、並列数を数えていました。「数十と数百では、並列化の方法や性能向上で注意しなくてはならないポイントは異なる」、などという点です。ところが、世界最高速の計算機では、万や数十万の並列数になってきており、アルゴリズムや考え方をすべて変える必要が出てき、これは、まったく新しいプログラムを開発するチャンスです。

仕事で使用する最も重要なツールとその理由

PC と高速インターネット接続です。それさえあれば、世界中どこにいても普段通りの仕事ができます。

開発初心者へのアドバイス

どのような単純なことでも、疑問があれば、自分で実装して確認することをお勧めします。インターネットで探した結果を鵜呑みにするような技術者ほど、想像力がなく、疑問さえ感じないようです。どのような技術でも原理があり、それらが深く絡み合って複雑な結果を生み出しています。新しい技術に貢献するには、自分で原理を見つけ、展開していかなくてはなりません。

インテル社で働く魅力

インテル社内の新しい技術を創造するたくさんの素晴らしい人たちと仕事ができるのは刺激的です。技術そのものが面白いですし、その技術のアプリケーションに貢献できること自体がぞくぞくするほど楽しい作業です。

余暇の過ごし方

自然の中でゆっくりと過ごすのが好きです。毎日 1 万歩歩き、毎週ゆっくりと 1km 泳ぎます。週末には自転車で山を登ったり、カヤックで川を下ったりします。目に入ってくる景色、耳で聞こえる風や水の音を感じながら、体を動かすとリラックスできます。汗をかいた後、温泉に入って湯上りの生ビールが飲めれば、もう最高です。

最適化に関する注意事項

インテル® コンパイラー、関連ライブラリーおよび関連開発ツールには、インテル製マイクロプロセッサーおよび互換マイクロプロセッサーで利用可能な命令セット（SIMD 命令セットなど）向けの最適化オプションが含まれているか、あるいはオプションを利用している可能性がありますが、両者では結果が異なります。また、インテル® コンパイラー用の特定のコンパイラー・オプション（インテル® マイクロアーキテクチャーに非固有のオプションを含む）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。これらのコンパイラー・オプションと関連する命令セットおよび特定のマイクロプロセッサーの詳細は、『インテル® コンパイラー・ユーザー・リファレンス・ガイド』の「コンパイラー・オプション」を参照してください。インテル® コンパイラー製品のライブラリー・ルーチンの多くは、互換マイクロプロセッサーよりもインテル製マイクロプロセッサーでより高度に最適化されます。インテル® コンパイラー製品のライブラリー・ルーチンの多くは、互換マイクロプロセッサーよりもインテル製マイクロプロセッサーでより高度に最適化されます。インテル® コンパイラー製品のコンパイラーとライブラリーは、選択されたオプション、コード、およびその他の要因に基づいてインテル製マイクロプロセッサーおよび互換マイクロプロセッサー向けに最適化されますが、インテル製マイクロプロセッサーにおいてより優れたパフォーマンスが得られる傾向にあります。

インテル® コンパイラー、関連ライブラリーおよび関連開発ツールは、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。

インテルでは、インテル® コンパイラーおよびライブラリーがインテル製マイクロプロセッサーおよび互換マイクロプロセッサーにおいて、優れたパフォーマンスを引き出すのに役立つ選択肢であると信じておりますが、お客様の要件に最適なコンパイラーを選択いただくよう、他のコンパイラーの評価を行うことを推奨しています。インテルでは、あらゆるコンパイラーやライブラリーで優れたパフォーマンスが引き出され、お客様のビジネスの成功のお役に立ちたいと願っております。お気づきの点がございましたら、お知らせください。

改訂 #20110307

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.co.jp/jp/software/products/