THE PARALLEL UNIVERSE

intel Software

ベクトル化特集

OpenMP* 向け SIMD ベクトル拡張

インテル® Advisor XE:
新しいベクトル化アドバイザー

Issue
22
2015

目次

編集者からのメッセージ

James Reinders インテル コーポレーションの並列プログラミング・エバンジェリスト兼ディレクター。
新しい書籍『High Performance Parallel Programming Pearls Volume Two』の共著者で、このほかに、『High Performance Parallel Programming Pearls (Volume One)』(2014)、『Multithreading for Visual Effects』(2014)、『Intel® Xeon Phi™ Coprocessor High Performance Programming』(2013、日本語 :『インテル® Xeon Phi™ コプロセッサー ハイパフォーマンス・プログラミング』)、『Structured Parallel Programming』(2012、日本語 :『構造化並列プログラミング』)、『Intel® Threading Building Blocks: Outfitting C++ for Multicore Processor Parallelism』(2007、日本語 :『インテル スレッディング・ビルディング・ブロック ―マルチコア時代の C++ 並列プログラミング』、中国語、韓国語の翻訳版があります)、『VTune™ Performance Analyzer Essentials』(2005) などの文献を発表しています。

ベクトル化で生き残る

ベクトル化というキーワードは、最近、私の中で大きなウェイトを占めています。それには、「ベクトル化アドバイザー」と呼ばれる新しいツールが登場したこと、OpenMP*「**SIMD**」宣言子をより深く理解したこと、**実世界のベクトル化例を含む新しい書籍**を執筆したことなど、さまざまな理由があります。

ベクトル化は、この春に「**Vectorize or Die (ベクトル化か死か)**」という名前で開催された 2 つのインテル® ソフトウェア・テクニカル Web セミナーのトピックでした。この言葉の意味は、簡単な計算を行うことですぐに理解できます。例えば、最新の並列 x86 製品であるインテル® Xeon Phi™ コプロセッサー (最大 244 スレッド、512 ビット幅ベクトル) で考えてみましょう。単純な計算では、244 スレッドを使用するスレッド化プログラムのスピードアップは 244 倍を超えることはありません。[1] また、完璧にベクトル化されたプログラムのスピードアップは 16 倍を超えることはありません (単精度、512 ビット SIMD)。しかし、これらを組み合わせて使用すると、「超えられない」スピードアップは 3,904 倍に跳ね上がります。つまり、スレッド化とベクトル化を組み合わせることが、パフォーマンス向上の鍵なのです。ベクトル化が最近話題になっている理由は、まさにここにあります。

Dr. Fortran (Steve Lionel 氏) は、Fortran ユーザーの調査に熱心に取り組み、ブログで紹介しています (**ページ 24 を参照**)。私はかねてからの Dr. Fortran のファンであり、彼の最新のブログは一見の価値があります。

この号の注目記事「OpenMP* SIMD によるベクトル・プログラミング ― インテル® Xeon® プロセッサー、インテル® Xeon Phi™ コプロセッサー、インテル® GPU 向け」では、OpenMP* で利用可能になった明示的なベクトル・プログラミング用の SIMD 宣言子を詳しく説明します。OpenMP* を使用しない場合でも、この記事の**明示的な**ベクトル・プログラミングに関する考察に目を通してみてください。明示的なベクトル化は、シーケンシャル・コードに基づいてベクトル化を行うため、これまでコンパイラーの判断によるベクトル化機能と、コンパイラーに与えることができる「ヒント」に依存していました。しかし、ベクトル化できなかったときにどのようなヒントやコンパイラー・オプションを追加すべきかは悩みの種でした。明示的なベクトル化によりベクトル化を直接プログラムすることができますが、コンパイラーの判断を覆す操作を行うことはコンパイラーによる保護を放棄することになります。詳細は記事をご覧ください。

「ベクトル化アドバイザー : ベクトル化を支援する新しいツール」では、ベクトル化を行う方法に関する質問への回答を提供する新しいツールを取り上げます。ベクトル化アドバイザーは、最新のインテル® Advisor XE の新機能です。この記事をお読みになった後、評価版をダウンロードしてコードを適切にベクトル化できるかどうか確認してみてください。

マルチスレッド・アプリケーションをベクトル化することで、ハイパフォーマンスを達成できるチャンスが大きく広がります。この号が効率的なベクトル化を達成する実践的な方法を見つけるのに役立つことを期待しています。

James Reinders
2015 年 9 月

1. ここでは、線形 (リニア) スピードアップを上回る、キャッシュ効果による超線形 (スーパーリニア) スピードアップは考慮していません。考慮しても数字が微妙に異なるだけで結論が異なることはないため、たやすく計算できる数値を取り上げています。

OpenMP* SIMD によるベクトル・プログラミング

インテル® Xeon® プロセッサー、インテル® Xeon Phi™ コプロセッサー、インテル® GPU 向け

Xinmin Tian インテル コーポレーション 主任エンジニア、Hideki Saito 主任エンジニア、Milind Girkar シニア主任エンジニア、Serguei Preis 主任エンジニア、Sergey Kozhukhov コンパイラー・エンジニア、Alejandro Duran ソフトウェア・エンジニア

最近の CPU や GPU における **SIMD** (Single Instruction Multiple Data) 実行ユニットのオンダイ統合は、消費電力 / 性能比を追求するために、SIMD ハードウェアを高度に活用するという課題を投げかけます。インテル® Xeon® プロセッサーとインテル® Xeon Phi™ コプロセッサーは、豊富な SIMD 命令セット (ギャザー / スキャッター、シャッフル、FMA、置換など) と、より幅の広いベクトルレジスター (256 ビットのインテル® AVX/ インテル® AVX2、512 ビットのインテル® MIC/ インテル® AVX512) を提供します。SIMD ハードウェア向けコンパイルの課題はプログラマーによるヒントで解決できますが、SIMD 並列処理を表現する業界標準の方法はありませんでした。このため、プログラマーは各コンパイラー・ベンダーによって提供されるベクトル並列処理向けの独自のヒントを利用するか、コンパイル時にプログラム要因により制限されるコンパイラーの自動**ベクトル化**機能を利用していました。

この状況を打開するため、OpenMP* 言語標準化委員会は、インテルの提案に基づいて、ベクトル・プログラミングをサポートする SIMD 構文を OpenMP* に追加しました。新しく標準化された構文により、プログラマーは、可搬性のないベンダー固有のベクトル化組込み関数や宣言子を使用する必要がなくなります。さらに、これらの SIMD 構文は、コンパイラーに対してコード構文に関する追加の情報を提供し、並列化と調和した、より優れたベクトル化を考慮しています。

この記事では、OpenMP* 4.0 仕様で利用可能になった、明示的なベクトル・プログラミング用の C/C++/Fortran SIMD 拡張について説明します。まず、単純な例を用いて、SIMD 構文と節のセマンティクスを説明します。続いて、プログラマーが効率的な SIMD プログラムを記述できるように、明示的なベクトル・プログラミングのガイドラインとプログラミング例を紹介します。ケーススタディーでは、インテル® Xeon Phi™ コプロセッサー上で OpenMP* 4.0 の PARALLEL および SIMD 構文を使用して約 2,000 倍のスピードアップを達成する方法を紹介します。また、インテル® Xeon® プロセッサー E3-1270 (開発コード名 Haswell) システムで SIMD 言語拡張を使用したワークロードで 2.28 倍から 6.67 倍のスピードアップを達成する方法についても紹介します。

BLOG HIGHLIGHTS

すべての責任はコンパイラーにあり

ANDREY KARPOV >

多くのプログラマーは、問題はコンパイラーが解決すべきであると考えがちです。この件について話すことにしましょう。これは本当でしょうか？ 実は、コンパイラーが問題を引き起こしているとプログラマーが提起した場合、その 99 パーセントは誤解によるものです。問題を調査すると、多くの場合、次のような原因が見つかります。

- **配列オーバーラン**
- **初期化されていない変数**
- **入力ミス**
- **並列プログラムの同期エラー**
- **volatile でない変数の使用**
- **不定な動作になるコード**

プログラマーは、この手の問題を修正したことがあるはずです。また、そういった情報を見たこともあるでしょう。しかし、すべての責任はコンパイラーにあると考えるプログラマーは少なくありません。責任がコンパイラーにあるように見えてしまうのです。

もちろん、実際に責任がコンパイラーにある可能性もあります。しかし、特殊なコンパイラーをマイクロコントローラー向けに使用するようなケースを除けば、その可能性は非常に低いものです。私は長年 Visual C++* で多くの作業を行っていますが、正しくないアセンブリー・コードが生成されたのは、これまで一度のみです。

この記事の続きはこちら (英語) でご覧になれます。 >

明示的なベクトル・プログラミングの理論的根拠

ループの**ベクトル化**は、最適化コンパイラーがベクトルレベルの並列処理を行う最も一般的な方法の 1 つです。[1, 2, 3, 4, 9, 10] しかし、ループをベクトル化する場合、**SIMD** ハードウェア上で実行するためすべての関数をインライン展開することなく関数呼び出しを制御する方法が問題となります。この問題は、**図 1** のように、ループにユーザー定義関数 **seqfun()** への呼び出しが含まれる場合に発生します。呼び出し元で関数がインライン展開されない限り、コンパイラーは **seqfun()** が何を行っているか分からないため、**図 1** のように、コンパイラーは呼び出し元のループを自動的にベクトル化しません。

```
extern float seqfun(float);
void callseqfoo(float *restrict a, float *restrict x, int n)
{ int i;
   for (i=0; i<n; i++) a[i] = seqfun(x[i]);
}
   sh-4.1$ icc -nologo -c -restrict -opt-report-phase:vec -opt-report-file=st
   derr vecfun.c
   最適化レポート開始: callseqfun(float * restrict , float *__restrict__, int)
   レポート:
   ベクトルの最適化 [vec]
   ループの開始 vecfun.c(4,4)
         リマーク #15543: ループはベクトル化されませんでした: 関数呼び出しを
         含むループは最適化の候補とは見なされません。 [ vecfun.c(4,31) ]
   ループの終了
```

1 ベクトル化されないループとユーザー定義関数の例

この問題に対処するため、OpenMP* ではプログラマーが declare SIMD 宣言子を使用してベクトル化できるユーザー定義関数に注釈を付けることができます。コンパイラーは、これらの関数に対する注釈を解析して処理します。これにより、**図 2** のように、プログラマーはベクトル化できると予想されるループと関数の両方をマークすることができます。

```
#pragma omp declare simd notinbranch
extern float seqfun(float) {
   // 関数本体
}
void callseqfun(float *restrict a, float *restrict x, int n)
{ #pragma omp simd
   for (int i=0; i<n; i++) a[i] = seqfun(x[i]);
}

sh-4.1$ icc -qopenmp -nologo -c -restrict -opt-report-phase:vec -opt-report-
file=stderr vecfun.c
   最適化レポート開始: seqfun..xN4v(float)
         レポート: ベクトルの最適化 [vec]
   リマーク #15301: 関数がベクトル化されました  [ vecfun.c(3,1) ]

   最適化レポート開始: callseqfun(float * restrict ,
   float * restrict , int) レポート:
         ベクトルの最適化 [vec]
   ループの開始 vecfun.c(10,4)
         リマーク #15301: OpenMP* SIMD ループがベクトル化されました
   ループの終了
```

2 ベクトル化されたループとユーザー定義関数の例

関数 **callseqfun()** と **seqfun()** は、同じコンパイル単位 (または同じファイル) に存在する必要はありません。しかし、関数 **callseqfun()** を declare SIMD をサポートするコンパイラーでコンパイルした場合、SIMD 実行を行うには、スカラー関数 **seqfun()** も同様に declare SIMD をサポートするコンパイラーでコンパイルする必要があります。**callseqfun()** のベクトル化により **vecfun()** の呼び出し (つまり、SIMD 化された **seqfun()**) が作成されます。また、注釈の付いた **seqfun()** のコンパイルには、ベクトルバージョンの **vecfun()** をオリジナルのスカラーバージョンの **seqfun()** に加えて生成する必要があります。SIMD ベクトル拡張を使用すると、一般的なループの入れ子のみ (またはループのみ) のベクトル化を超えた、関数のベクトル化が可能になります。

シリアル実行				SIMD 実行
反復-0	反復-1	反復-2	反復-3	ベクトル反復-0
call seqfun(x[0]) -> r0	call seqfun(x[1]) -> r1	call seqfun(x[2]) -> r2	call seqfun(x[3]) -> r3	call vecfun(x[0…3]) -> vecreg0
store a[0], r0	store a[1], r1	store a[2], r2	store a[3], r3	simdstore a[0…3], vecreg0

3 スカラー関数をベクトル関数に変換して SIMD ハードウェアで実行する例

図 3 のように、**callseqfun()** の **omp simd** ループをベクトル化する際、コンパイラーのコード変換によりスカラー関数呼び出し **float seqfun(float x)** はベクトル関数呼び出し **_m128 vecfun(_m128 vecreg)** に変更されます。基本的に、コンパイラーはオリジナルのスカラー関数とベクトル注釈に基づくベクトルバージョンの関数を生成します。ベクトル化の後、呼び出し元の 4 つのスカラー関数呼び出し (**seqfun(x0)**、**seqfun(x1)**、**seqfun(x2)**、**seqfun(x3)**) は、ベクトルレジスターを使用する 1 つのベクトル関数呼び出し **vecfun(<x0…x3>)** (ベクトル入力引数 **<x0…x3>** およびベクトル戻り値 **vecreg0 = <r0…r3>**) に置換されます。入力引数 x0、x1、x2、 x3 は 1 つの 128 ビット・ベクトル・レジスターにパックして渡し、結果 r0、r1、r2、r3 は 1 つの 128 ビット・ベクトル・レジスターにパックして返すことができます。**vecfun()** の実行はベクトルレベルの並列処理を適用できます。

OpenMP* ループ構文で `for` (C/C++) または `do` (Fortran) ループのスレッドレベルの並列処理を定義しているにもかかわらず明示的なベクトル・プログラミングを行う必要がある理論的根拠は、明示的なベクトル・プログラミングにおいて既存の構文を再利用することが難しいためです。**図 4** の例は、(部分的に) 並列で実行できるが、ベクトル化および SIMD ベクトル長選択の安全性に関する追加の情報がないとコンパイラーがベクトル化できないコードを示しています。

```
#define N 1000000
float x[N][N], y[N][N];
#pragma omp parallel for
for (int i=0; i<N; i++) {
   #pragma omp simd safelen(18)
   for (int j=18; j<N−18; j++) {
         x[i][j] = x[i][j−18] + expf(y[i][j]);
         y[i][j] = y[i][j+18] + logf(x[i][j]);
   }
}
```

4 ワークシェアリング OMP for と OMP **SIMD** 構文を使用した例

図 4 には、ループ (**`for`**) 構文でのみベクトル化を妨げる 2 つの問題があります。

- ループ構文のセマンティクスに違反するため、ループ構文は j ループに拡張できません。
- j ループには、ベクトル化を妨げる後方へのループ伝搬依存性が含まれています。

しかし、配列 **`y`** のベクトル長が明示され、配列 **`x`** のベクトルが 18 要素よりも短い場合、コードはベクトル化できます。これらの問題に対処するため、構文と節が OpenMP* に追加され、プログラマーは並列領域とループに加えて SIMD 関数とループに明示的に注釈を付けてベクトルレベルの並列処理を利用できるようになりました。

SIMD ベクトル言語拡張

明示的なベクトル・プログラミングを促進するため、新しいプラグマ (宣言子) が OpenMP* 4.0 仕様に追加されました。[8] 注釈付きの C/**C++** および Fortran ループと関数は、最新のマイクロプロセッサーで SIMD 実行が可能であるため、プログラマーはコンパイラーのデータ依存性解析やベンダー固有のコンパイラー・ヒント (例えば、IBM*、Cray*、インテル® コンパイラーでサポートされている **#pragma ivdep**) を指定する必要はありません。

このセクションでは、これらの拡張の構文とセマンティクスを説明します。OpenMP* SIMD 拡張には制限があります。例えば、C++ 例外処理コード、例外をスローする呼び出し、**longjmp** および **setjmp** 関数呼び出しは、SIMD 関数とループ構文または動的スコープでは使用できません。このほかに、次のような制限があります。

- 関数やサブルーチンの本体は構造化ブロックでなければいけません。
- SIMD ループから呼び出された場合、関数やサブルーチンを実行しても OpenMP* 構文は実行されません。
- 関数やサブルーチンの実行で、SIMD チャンクの同時反復が変更されてはいけません。
- 関数へまたは関数から分岐するプログラムは非準拠です。

OpenMP* SIMD 拡張の詳細な構文の制限と言語規則は、OpenMP* 4.0 または OpenMP* 4.1 (ドラフトバージョン) の仕様を参照してください。[8]

ループの SIMD 構文

OpenMP* 4.0 SIMD 拡張の基本は、次の図のように、`for` (C/C++) ループおよび `do` (Fortran) ループの SIMD 構文です。この新しい構文は、ループのベクトル化をコンパイラーに指示します。SIMD 構文は、ループの反復を特定の長さの連続するチャンクに分割できることを示します。各チャンクは、オリジナルのシリアルプログラムおよびその実行のすべてのデータ依存性を保持しながら、チャンク内の複数の SIMD レーンで同時に実行することができます。SIMD 構文のシンタックスは次のとおりです。

```
C/C++:
  #pragma omp simd [clause[[,] clause] ...] new-line
  for-loops
Fortran:
  !$omp simd [clause[[,] clause] ...] new-line
  do-loops
  [!$omp end simd ]
```

SIMD 構文は、既存のループ構文の概念とシンタックスに従います。後述の「SIMD 構文の節」セクションで説明している節をサポートします。関連する for または do ループのループヘッダーは、ループ構文については同じ制限に従います。OpenMP* コンパイラーは、これらの制限によりループの反復空間を決定し、適切にベクトル化されるように分散します。

SIMD 構文を既存のワークシェアリング・ループ構文 (および並列ループ) に適用して、SIMD 命令を同時に実行可能なループを指定する、ループ SIMD 構文 (および結合並列ループ SIMD 構文) を形成することもできます。それらの反復もチームのスレッドで並列実行されます。

ループ (for または do) SIMD 構文は、最初に、ループ構文に適用される節に応じて、並列領域の暗黙的なタスクに関連するループの反復を分散します。そして、生成された反復のチャンクは、SIMD 構文に適用される節に応じて SIMD ループに変換されます。あたかも適用されていたかのように両方の構文に適用される節の効果があります。詳細は、OpenMP* 4.0 仕様のセクション 2.8.3 およびセクション 2.10 を参照してください。

関数の SIMD 構文

ループの SIMD 構文に加えて、OpenMP* 4.0 では declare SIMD 構文が追加されました。この構文は、関数 (C、C++、Fortran) またはサブルーチン (Fortran) に適用して、SIMD ループの単一呼び出しから SIMD 命令を利用し、各引数の複数のインスタンスを処理する 1 つ以上のバージョンを作成できます。関数 (C、C++、Fortran) またはサブルーチン (Fortran) 用に複数の declare SIMD 宣言子があります。declare SIMD の構文は次のとおりです。

```
C/C++:
  #pragma omp declare simd [clause[[,] clause] ...] new-line
  [#pragma omp declare simd [clause[[,] clause] ...] new-line]
  function definition or declaration
Fortran:
  $omp declare simd (proc-name) [clause[[,] clause] ...] new-line
```

declare SIMD 構文は、関連する関数の SIMD バージョンを作成するようにコンパイラーに指示します。この宣言子の節で指定する式は、関数宣言または関数定義における引数のスコープで評価されます。

SIMD 実行モデル

SIMD ループには、0、1、...、*N*-1 と番号付けされた論理的な反復が含まれます。ここで、*N* はループの反復回数です。論理的な番号付けは、関連するループが SIMD 命令なしで実行された場合に反復が実行される順番を示します。SIMD 関数には、0、1、…、*VL*-1 と番号付けされた論理的な番号が含まれます。ここで、*VL* は SIMD レーンの数です。論理的な番号付けは、関連する関数が SIMD 命令なしで実行された場合に呼び出しが実行される順番を示します。つまり、正当な SIMD プログラムとその実行は、反復 (または呼び出し) 間のすべてのオリジナルのデータ依存性と、シリアルプログラムとその実行の反復内の依存性に従うべきです。

反復のチャンクは SIMD レーンにマップされ、それらの SIMD レーンで同時に実行を開始します。実行する SIMD レーンのグループは、**SIMD チャンク**と呼ばれます。1 つのプログラムカウンターがすべての SIMD レーンにより共有され、次に実行される単一の命令を指します。SIMD チャンク内の実行を制御するため、実行プレディケートという現在の命令の副作用を識別すべきかどうかを示す SIMD レーンごとのブール値があります。例えば、文が「すべて偽」プレディケートで実行された場合、識別できる副作用があってはいけません。アプリケーションの SIMD コンテキスト (SIMD ループや SIMD 関数) の入口で、`for` SIMD ループと notinbranch SIMD 関数の実行プレディケートは「すべて真」で、プログラムカウンターはループまたは関数の最初の文を指します。次の 2 つの文は、SIMD コンテキスト実行中のプログラムカウンターと実行プレディケートの動作を説明しています。

> プログラムカウンターは、SIMD コンテキストの文と文の間の保守的な実行パスに対応する値のシーケンスです。いずれかの SIMD レーンが文を実行する場合、プログラムカウンターはその文を渡します。
>
> プログラムカウンターが文を渡す際に、実行プレディケートはその文を実行する SIMD レーンの値のみ「真」に設定されます。

上記の SIMD 実行の動作はコンパイラーに一定の判断の自由度を与えます。例えば、プログラムカウンターは、文に識別できる副作用がないために実行プレディケートが「すべて偽」である一連の文をスキップすることができます。実際には、プログラムの制御フローは分岐する場合があります。その場合、異なる SIMD レーンが異なる計算を実行する必要があるため SIMD の効率 (およびパフォーマンス) は低下します。SIMD 実行モデルは、「SIMD レーンの実行は**最大限に収束する**」というプログラムカウンターと実行プレディケートに関する重要な保証を提供します。最大限の収束とは、2 つの SIMD レーンが同じ制御パスを実行する場合、すべてのオリジナルのデータ依存性を保持したまま各プログラム文を同時に実行することが保証されることを意味します。2 つの SIMD レーンが分岐する制御パスを実行する場合は、SIMD 実行でできるだけ早く再収束することが保証されます。制御フローの合流点で、分岐した SIMD 実行を統合して、1 つの実行モードのように再び継続することができます。

SIMD 構文の節

SIMD 構文の実行動作を細かく制御するため、OpenMP* では、プログラマーがデータ共有、データ移動、可視性、SIMD 長、線形性、均一性、メモリー・アライメントを明示的に指定して最適なベクトルプログラムを記述できる、さまざまな節が提供されています。

データ共有節

private、lastprivate、reduction 節は、データのプライベート化および SIMD 実行コンテキストの変数の共有を制御します。private 節は、指定された変数の初期化されていないプライベートのベクトルを作成します。SIMD 関数の引数は、デフォルトで、ハードウェア SIMD レーン間のインスタンスごとに、個別の格納場所または値を持ちます (つまり、プライベートです)。lastprivate 節は、同じセマンティクスを提供するだけでなく、最後の反復で生成された値をループの外部にコピーします。reduction 節は、変数のベクトルコピーを作成して、そのベクトルの部分的な値をオリジナルのスカラー変数にマージします。

uniform 節。uniform 節で指定されるパラメーターは、単一 SIMD ループの実行における関数の同時呼び出しのチャンクの不変値を表し、ベクトル実行の SIMD レーン間で共有されます。SIMD レーン間で共有される関数パラメーターを均一として指定すると、ベクトライザーはスカラー (またはユニットストライド) メモリーロード (またはストア) 用に最適化されたコードと最適な制御フローを生成することができます。例えば、ベースアドレスが均一で、オフセットが線形ユニットストライドの場合、コンパイラーは、ギャザー / スキャッター命令を生成する代わりに、より高速なユニットストライド方式のベクトル・メモリー・ロード / ストア命令 (インテルの SIMD ハードウェアでサポートされている movaps や movups など) を生成することができます。また、制御フロー決定のテスト条件が一定の量に基づく場合、コンパイラーは、マスクのチェックと制御フロー発散のオーバーヘッドを抑えるために、すべてのインスタンスが同じパスを実行するようにできます。

linear 節。linear 節で指定される変数 (またはパラメーター) は、各反復でプライベートで、SIMD 実行コンテキストの反復空間と線形の関係があります。変数は、2 つ以上の linear 節、または 1 つの linear 節と別の OpenMP* データ節で使用することはできません。linear 節の変数では、線形ステップを指定できます。指定する場合、線形ステップ式は構文に関連する領域の実行中は不変でなければなりません。不変でない場合、予期しない動作が発生することがあります。線形ステップを指定しない場合は、1 であると仮定されます。

SIMD ループ・コンテキストでは、関連するループの各反復の線形化された変数値は、(構文に入る前のオリジナルの変数値) ＋ (論理的な反復回数) × (線形ステップ) に相当します。関連するループの順序的に最後の反復に相当する値は、オリジナルの変数に割り当てられます。関連するコードがループの各反復の線形ステップで変数を増加させない場合、動作は不定です。

aligned 節。多くのプラットフォームはアライメントされたデータをアライメントされていないデータよりも高速にロード (またはストア) できるため、特に SIMD (ベクトル型) データでは、メモリーアクセスのアライメントは重要です。しかし、コンパイラーは、プログラムのすべてのモジュールにわたるデータのアライメント特性や動的に割り当てられたメモリー (またはオブジェクト) を検出できないことがあるため、アライメントされていないロード / ストアのみを使用する保守的なコードを生成します。プログラマーは、**`aligned(variable[:alignment][,variable[:alignment]])`** 節を使用することで、コンパイラーにアライメント情報 (正の整数値のバイト数など) を伝えることができます。プログラマーは、aligned 節のリストの各変数についてアライメント値を指定できます。オプションのアライメント値が指定されていない場合、ターゲット・プラットフォームの SIMD 命令用の実装で定義されるデフォルトのアライメントが仮定されます。

safelen 節。safelen 節を指定すると、SIMD 命令で同時に実行される 2 つの反復は、論理的な反復空間でその値より大きな距離を持つことはできません。safelen 節のパラメーターは正の整数式です。同時に実行される反復数は実装で定義されますが、safelen 節で指定した値を超えないことが保証されます。個々の SIMD レーンはそれぞれ同時反復を実行します。同時反復の各セットは SIMD チャンクです。

simdlen 節。SIMD バージョンの作成時に **`declare simd`** で注釈が付けられた関数については、関数の各引数にパックされる同時要素の数は、simdlen 節で指定されたベクトル長により決定されます (デフォルトでは、指定した SIMD ハードウェア向けにコンパイラーにより選択されます)。指定されたベクトル長がハードウェア SIMD 長の倍数の場合、コンパイラーは、複数のベクトルレジスターをより大きな論理ベクトルレジスターに融合してより長いベクトルをエミュレートする double-pumping (2 倍速)、triple-pumping (3 倍速)、quad-pumping (4 倍速) を適用します。 simdlen 節のパラメーターは正の整数式です。実際には、ハードウェア SIMD 長の倍数にします。倍数にしない場合、関数の各引数でパックされる要素の数は実装で定義されます。

inbranch 節と notinbranch 節。inbranch 節は、関数が常に SIMD ループ / 関数の条件の下で呼び出されることを示します。notinbranch 節は、関数が SIMD ループ / 関数の条件の下で呼び出されないことを示します。どちらの節も指定されない場合、関数は SIMD ループ / 関数の条件文の内部から呼び出されることもあれば、呼び出されないこともあります。デフォルトでは、宣言されたすべての異なる SIMD 関数について、プレディケートを含む条件付き呼び出し用の実装 (inbranch バージョン) と無条件呼び出し用の実装 (notinbranch バージョン) の 2 つの実装が提供されます。

すべての呼び出しが条件付きの場合、inbranch 節を使用すると notinbranch バージョンは生成されません。同様に、すべての呼び出しが無条件の場合、notinbranch 節を使用すると inbranch バージョンは生成されません。SIMD 関数の inbranch または notinbranch バージョンを生成しないと、コードサイズが減少し、コンパイル時間が短縮されます。デフォルトでは、コンパイラーはオリジナルのスカラー関数が条件の下で常に呼び出されるかどうか判断できないため、inbranch バージョンと notinbranch バージョンの両方を生成します。

processor 節。インテルのインテル® アーキテクチャー用 SIMD 拡張である processor 節は、オリジナルのスカラー関数と指定されたターゲット・プロセッサー用のベクトル注釈に基づいてベクトルバージョンの関数を作成するようにコンパイラーに指示します。デフォルトのプロセッサー (または ISA クラス : XMM、YMM、ZMM) の選択は、ベクトル ABI (アプリケーション・バイナリー・インターフェイス) 仕様に基づいて行われます。[2] コンパイラーのコマンドラインで暗黙的または明示的なプロセッサー固有 / アーキテクチャー固有のオプション (例えば、Linux* でインテル® コンパイラーの –xSSE4.2 オプション) を指定して ISA クラス選択を制御することはできません。ターゲット processor 節は、コンパイラーが使用できるベクトル ISA とベクトルの幅の両方を指定します。

```
processor-clause: processor ( processor-name )
processor-name:
   pentium_4
   pentium_4_sse3
   … … … …
   core_i7_sse_4_2    (SSE4.2)
   core_4th_gen_avx   (Haswell)
```

実装の注意点 : すべてのコンパイラーおよびハードウェア・ベンダーは独自のプロセッサー名を定義してサポートできます。

合成ループ SIMD 構文

ループ SIMD 構文 (例えば、**#pragma omp for simd**) は、ワークシェアリング・ループと SIMD ループの合成構文[8] です。最初に、1 つ以上の関連するループの反復がチームにすでに存在するスレッド間で分散され、次に、各スレッドに分配された反復が SIMD 構文に適用される節に応じて SIMD ループに変換されます。節は、あたかもそれぞれの構文で指定されていたかのように両方の構文に適用されます (1 回のみ適用される collapse 節を除きます)。**図 5** の例を参照してください。

```
float a[M][N], b[M][N], c[M][N];
#pragma omp for simd collapse(2)
for (m=0; m<M; m++) {
   for (n=0; n<N; n++) {
   c[m][n] = c[m][n] + a[m][n] * b[m][n];
   }
}
```

5 合成ループ SIMD 構文の例

図 5 の二重の入れ子のループは、合成ループ SIMD 構文で指定されています。プログラマーは、このループの入れ子を複数のスレッドと SIMD 命令により同時に実行できることをコンパイラーに知らせています。collapse 節を適用すると、このループの入れ子は M*N 反復のループに変換され、並列の反復チャンクサイズおよび SIMD 実行は増加します。例えば、M=640 で N=16、スレッド数が 64 の場合、各スレッドの反復チャンクは 160 になります。インテル® AVX512 対応のインテル® Xeon® プロセッサーおよびインテル® Xeon Phi™ コプロセッサーは、この 160 の反復チャンクを、ベクトル長 16 の 10 のベクトル反復の SIMD ループとして実行します。**図 6** は、生成された疑似 SIMD コードを示しています。

```
......
simdized_for (k=simd_chunk_start; k<simd_chunk_start+160; k+=16) {
simd_c[k:16] = simd_c[k:16] + simd_a[k:16] * simd_b[k:16];
}
......
```

6 合成ループ SIMD 構文の疑似 SIMD コード

SIMD ベクトル・プログラミングのガイドライン

このセクションでは、SIMD 拡張を使用して正しいハイパフォーマンスなベクトルプログラムを記述するためのガイドラインを紹介します。

SIMD 実行の正当性の保証

プログラマーが SIMD 構文をループに適用して SIMD ループに変換する場合、ループがチャンクに分割され、各チャンク内の反復が SIMD 命令を使用して同時に正しく実行できることを保証する必要があります。この保証を行うには、safelen 節を使用して、すべてのオリジナルのデータ依存性を保持するか、private、lastprivate、reduction のようなデータ共有節を指定して SIMD 実行を妨げるデータ依存性を排除する必要があります。次のことを思い出してください。

- SIMD プラグマ / 宣言子で注釈が付けられたループには、0、1、...、*N*-1 と番号付けされた論理的な反復が含まれます。ここで、*N* はループの反復回数です。
- 論理的な番号付けは、関連するループが SIMD 命令なしで実行された場合に反復が実行される順番を示します。
- safelen(*L*) 節を指定すると、SIMD 命令で同時に実行される 2 つの反復は、論理的な反復空間で *L* より大きな距離を持つことはできません。

プログラマーが以下のプログラミング・ガイドラインに従うことで、OpenMP* の標準的なループが SIMD 実行用に正しく変換されることが保証されます。

ガイドライン 1: 論理的なループ反復では、safelen 節を使用して、チャンクの 2 つの反復間で後方へのループ伝搬依存性を禁止します。例えば、チャンクが *[k, k+1, k+2, k+3]* の場合、文 *S* の反復 *k* により生成される結果は、文 *S* の反復 *k+1*、*k+2*、*k+3* または *S* よりも上の文で使用されてはいけません。

```
#pragma omp simd safelen(5)
for (k=5; k<N; k++) {
a[k] = a[k-5] + b[k];
}
```

7 safelen 節を使用した SIMD ループの例

図 7 の safelen 節は、ベクトル長 (VL) が 5 未満の場合に安全に SIMD ループに変換されると仮定しています。コンパイラーが VL=4 を選択した場合の SIMD 実行を次に示します。反復-n はシリアル実行の論理的な反復番号、ベクトル反復-n は SIMD 実行の論理的なベクトル反復番号、*r-n* は一般的なスカラーレジスター、*vr-n* はベクトルレジスターをそれぞれ示します。

シリアル実行				SIMD 実行
反復-0	反復-1	反復-2	反復-3	ベクトル反復-0
load r0, a[0]	load r0, a[1]	load r0, a[2]	load r0, a[3]	simdload vr0, a[0…3]
load r1, b[5]	load r1, b[6]	load r1, b[7]	load r1, b[8]	simdload vr1, b[5…8]
add r0, r1	add r0, r1	add r0, r1	add r0, r1	simdadd vr0, vr1
store a[5], r0	store a[6], r0	store a[7], r0	store a[8], r0	simdstore a[5…8], vr0
反復-4	反復-5	反復-6	反復-7	ベクトル反復-1
load r0, a[4]	load r0, a[5]	load r0, a[6]	load r0, a[7]	simdload vr0, a[4…7]
load r1, b[9]	load r1, b[10]	load r1, b[11]	load r1, b[12]	simdload vr1, b[9…12]
add r0, r1	add r0, r1	add r0, r1	add r0, r1	simdadd vr0, vr1
store a[9], r0	store a[10], r0	store a[11], r0	store a[12], r0	simdstore a[9…12], vr0

8 safelen 節を使用した SIMD ループの実行サンプル

図 8 で、ループの論理的な反復のシリアル実行から、反復-0 *store a[5]* により生成される結果は反復-5 *load a[5]* に伝搬され、反復-1 *store a[6]* により生成される結果は反復 -6 *load a[6]* に伝搬されます (以下同様)。また、*store a[k]* から *load a[k-5]* の依存性の順序は逆です。つまり、このループには、反復-(k) と反復-(k+5) の間に後方へのループ伝搬依存性が含まれています (k=0、1、2、…、N-5)。そのため、このループは VL>5 ではベクトル化できません。プログラマーが safelen(5) を指定したため、**図 8** のベクトル反復-0 とベクトル反復-1 の SIMD 実行のように、コンパイラーはループを正しくベクトル化することができます。

ガイドライン 2: 論理的なループ反復空間では、データ共有節を使用してチャンクの 2 つの反復で *write-write* や *write-read* の競合がないことを保証します。次の例は、private 節を使用してこの保証を行う方法を示しています。

```
{ float w;
   #pragma omp simd private(w)
   for (k = 0; k<N; k++) {
         w = a[k];
         b[k] = foo(w * a[k+1]);
   }
}
```

9 private 節を使用した SIMD ループの例

図 9 では、変数 *w* をプライベート化することにより、ループは *VL<N* で安全にベクトル化されます。しかし、*w* が各 SIMD レーン用にプライベート化されていない場合、*w* を含む *write-write* および *write-read* の競合が発生します。

uniform 節と linear 節の効率的な利用

uniform 節は、すべての SIMD レーンで同じ値のパラメーター (つまり、均一値パラメーター) に使用します。ベクトルレジスターの代わりにスカラーレジスターを使用してパラメーター値 (ポインターの場合はアドレス) を渡すコードを生成するようにコンパイラーに指示します。

ガイドライン 3: uniform 節は、(均一値) パラメーターがメモリー参照の均一ベースアドレス計算の一部として使用された場合、または均一制御フローの評価で使用された場合 (つまり、スカラー値として使用された場合) に最も効果的です。ほかの多くのケースでは、uniform 節の使用を避けることにより、スカラーからベクトルへのブロードキャスト操作を呼び出し側ループの外にホイストする (巻き上げる) ことができます。よく考えてください。スカラー・パラメーター (または変数) の linear 節は、指定されたスカラー入力値から線形シーケンスを暗黙的または明示的に生成するコードの生成をコンパイラーに指示します。線形パラメーター値はベクトルレジスターの代わりにスカラーレジスターを使用して渡されます。

ガイドライン 4: SIMD ループ内で線形更新される変数には、必ず linear 節を使用する必要があります。必要な場合に linear 節を使用しないと、予期しない動作を引き起こすことがあります。

ガイドライン 5: 場合によっては、SIMD 関数のパラメーターに linear 節を使用すべきです。(線形値) パラメーターがベクトルメモリー参照アドレス計算の一部として使用された場合 (ユニットストライド方式のベクトルロード / ストアおよびストライド方式のベクトルロード / ストア) に最も効果的です。ほかの多くのケースでは、linear 節の使用を避けることにより、ベクトル・インダクション値の生成を呼び出し側でより効率的に行うことができます。

uniform/linear 節の一般的な利用例は、メモリーアクセスのベースアドレスが均一で、インデックス (またはオフセット) に線形特性が含まれる場合です。これらを組み合わせて、コンパイラーは、より優れたパフォーマンスが得られる線形ユニットストライド方式のメモリーロード / ストア命令を生成します。次の例では、関数 **SqrtMul** に **pragma omp declare simd** 注釈が付けられています。

```
#pragma omp declare simd uniform(op1) linear(k) notinbranch
processor(core_4th_gen_avx)
double SqrtMul(double *op1, double op2, int k)
{
   return (sqrt(op1[k]) * sqrt(op2));
}
```

10 uniform 節と linear 節を使用した SIMD ループの例

図 10 の例では、コンパイラーは、より高速な SIMD ベクトル関数のためにインテル® AVX2 対応の第 4 世代インテル® Core™ プロセッサー (開発コード名 Haswell) 向けの **SqrtMul** ベクトル関数を生成しています。コンパイラーは、**uniform(op1)** および **linear(k:1)** 属性を使用して、メモリーアドレス計算用に rax レジスターのベースアドレスと ecx レジスターの初期オフセット値を渡します。次に、**図 11** のように、**vsqrtpd** 命令を使用して、256 ビット YMM レジスターに格納する 4 つの 64 ビット浮動小数点データ (a[k]、a[k+1]、a[k+2]、a[k+3]) を計算します。

```
;; Generated vector function for
;; #pragma omp declare simd uniform(op1) linear(k:1) notinbranch
processor(core_4th_gen_avx)
   _ZGVYN4uvl_7SqrtMulPddi
   ; parameter 1: rax  ; uniform op1
   ; parameter 2: ymm0 ; ymm0 holds 4 values of op2_1, op2_2, op2_3 and op2_4
   ; parameter 3: ecx  ; linear k with unit stride
   movslq %ecx, %rcx
   vsqrtpd %ymm0, %ymm1    ; vector_sqrt(ymm0)      => ymm1
   vsqrtpd (%rax,%rcx,8),%ymm2 ; vector_sqrt(op1[k:3])    => ymm2 vmulpd
   %ymm2, %ymm1, %ymm0 ; vector multiply: ymm2 * ymm1 => ymm0 ret
```

11 uniform 節と linear 節を使用した SIMD ループ用に生成されたインテル® AVX2 コード

プログラマーが uniform 節と linear 節を省略した場合、コンパイラーは、すべての SIMD レーンのメモリーロード / ストアのベースアドレスが同じであり、オフセットが線形ユニットストライドであることを判断できません。そのため、**図 12** の例で示すように、コンパイラーは、SIMD 実行コンテキストでスカラー関数 **SqrtMul** に op1、op2、k を渡す際に YMM レジスターを使用する必要があります。メモリーアドレス計算では、コンパイラーは、ベクトル命令 **vsqrtpd** を使用して、**vector_sqrt(ymm7)** のベクトル実行を行うため、倍精度 op1_1[k1]、op1_2[k2]、op1_3[k3]、op1_4[k4] データを ymm7 レジスターにロードする SIMD ギャザー命令 vgatherqpd を生成します。呼び出し側の関数呼び出しで均一メモリーアドレス op1 および線形ユニットストライド値 *k* を渡す場合でも、このバージョンを呼び出すとパフォーマンスは大幅に低下します。

```
;; Generated vector function for #pragma omp declare simd notinbranch
processor(core_4th_gen_avx)
_ZGVYN4vvv_7SqrtMulPddi
; parameter 1: ymm0  ; vector_op1 holds op1_1, op1_2, op1_3 and op1_4
; parameter 2: ymm1  ; vector_op2 holds op2_1, op2_2, op2_3 and op2_4
; parameter 3: ymm2  ; vector_k  holds k1, k2, k3, k4
vpmovsxdq    %xmm2, %ymm3          ; save vector_k => ymm3
vpsllq       $3, %ymm3, %ymm4      ; vector_k*8 is index value => ymm4
vpaddq       %ymm0, %ymm4, %ymm5   ; vector_op1 + vector_k*8 => ymm5
vsqrtpd      %ymm1, %ymm0          ; vector_sqrt(ymm1) => ymm0
vpcmpeqd     %ymm6, %ymm6, %ymm6   ; set ymm6 (base) to zero for vgather
                                     instruction
vxorpd       %ymm7, %ymm7, %ymm7   ; clear up ymm7
vgatherqpd   %ymm6, (,%ymm5) %ymm7 ; vector_gather[op1_1[k1], op1_2[k2],
                                     op1_3[k3], op1_4[k4]
                                     => ymm7
vsqrtpd      %ymm7, %ymm1          ; vector_sqrt(ymm7) => ymm1
vmulpd       %ymm1, %ymm0, %ymm0   ; vector multiply: ymm1 * ymm0 => ymm0
ret
```

12 uniform 節と linear 節を使用しない SIMD ループ用に生成されたインテル® AVX2 コード

インテル® C/**C++ コンパイラー**を (インテル® Xeon® プロセッサー E3-1270 [開発コード名 Haswell]、64 ビット Windows* プラットフォームで) 使用し、**-QxCore-AVX2** および **-DSIMDOPT** オプションを指定してコンパイルした場合、**図 13** の uniform 節と linear 節を使用した小さなカーネルプログラム (呼び出し元関数と呼び出し先関数は異なるファイルで実装) から生成されたベクトルバージョンの関数は、節を省略した場合の 1.78 倍にスピードアップしました。

```
#include <stdio.h>
#include <stdlib.h>
#define M 1000000
#define N 1024

// ファイル: fSqrtMul.c void init(float a[])
{ int i;
   for (i = 0; i < N; i++) a[i] = (float)i*1.5;
}

float checksum(float b[])
{ int i; float res = 0.0;
   for (i = 0; i < N; i++) res += b[i]; return res;
}

#pragma omp declare simd simdlen(8) processor(core_4th_gen_avx)
#ifdef SIMDOPT
#pragam omp declare simd linear(op1) uniform(op2) simdlen(8) processor(core_4th_
gen_avx)
#endif
float fSqrtMul(float *op1, float op2) {
   return sqrt(*op1)*sqrt(op2);
}

// ファイル main.c
int main(int argc, char *argv[])
{ int i, k; float a[N], b[N]; float res = 0.0f; init(a);
for (i = 0; i < M; i++) {
   float op2 = 64.0f + i*1.0f;
#pragma omp simd
         for (k=0; k<N; k++) {
               b[k] = fSqrtMul(&a[k], op2);
         }
   }
   res = checksum(b); printf("res = %.2f\n", res);
}
```

13 uniform 節と linear 節を使用した例

ベクトル長の選択について

プログラマーが **simdlen(VL)** 節と **declare simd** 宣言子を使用してベクトル長 (VL) を直接指定しない場合、コンパイラーは、ターゲット命令セットの物理ベクトル幅とデータ型に基づいて VL を決定します。例えば、インテル® アーキテクチャーでは、次の規則に従って VL が選択されます。

- ターゲット・プロセッサーが XMM ベクトル・アーキテクチャーをサポートしていて (つまり、YMM ベクトルをサポートしないで) 関数のデータ型が **int** の場合、VL は 4 です。
- ターゲット・プロセッサーがインテル® アドバンスト・ベクトル・エクステンション (インテル® AVX) の YMM をサポートしていて関数のデータ型が **int** の場合、VL は 4 です (インテル® AVX の整数ベクトル演算は XMM で実行されます)。
- 関数のデータ型が **float** の場合、VL は 8 です。
- 関数のデータ型が **double** の場合、VL は 4 です。

多くのベクトルレジスターが必要ないアプリケーションでは、プログラムを倍のベクトル幅 (2 つの XMM レジスターを使用してインテル® SSE で 8 ワイド、2 つの YMM レジスターを使用してインテル® AVX で 16 ワイド) でコンパイルすると、パフォーマンスが向上します。この方法では、命令レベルで並列処理が行われ、プログラム・インスタンス上のさまざまなオーバーヘッドが相殺されることにより、大幅に実行が効率化されます。その他のワークロードでは、ベクトル幅が広くなるために処理が遅くなることがあります。重要なカーネルでは、simdlen 節を使用する両方のアプローチを試してみる価値があります。

メモリーアクセスのアライメント

アライメントの最適化は、最新の SIMD ハードウェアでは重要です。[9] 新しいアーキテクチャーではベクトルレジスターの幅が増える傾向にあるため、その重要性はますます高くなるでしょう。OpenMP* 4.0 では、**aligned(n)** 節を使用してアライメント情報を示すことができます。コンパイラーが最新のインテル® アーキテクチャー・ベースのシステムで最適な SIMD コードを生成するには、*n* を 8、16、32、64 にして 8 バイト、16 バイト、32 バイト、64 バイト・アライメントを指定します。

例えば、適切な配列要素アクセスのアライメントは、インテル® Pentium® 4 プロセッサーからインテル® Core™ i7 プロセッサーでは 16 バイト、インテル® AVX 対応プロセッサーとインテル® AVX2 対応プロセッサーでは 32 バイト、インテル® Xeon Phi™ コプロセッサーでは 64 バイトです。

```
void arrayref(float *restrict x, float *y, int n, int n1) {
_assume(n1%8=0);
#pragma omp simd aligned(y:32)
   for (int k=0; k<n; k++) {
   x[k] = x[k] + y[k] + y[k+n1] + y[k-n1];
   }
}
```

14 aligned 節を使用した SIMD ループの例

図 14 の例では、配列 *x* とポインター *y* はそれぞれ、32 バイト・アライメントとしてマークされています。メモリー参照オフセット *n1* は、`mod 8 = 0` でアサートされています。これらの注釈は、すべてのベクトルメモリー参照が 32 バイトでアライメントされていることをコンパイラーに知らせます。

構造体と多次元配列

スカラーおよびユニットストライド方式のメモリーアクセスのベクトル化は、SIMD アーキテクチャーおよび最近のコンパイラーでは効率良くサポートされています。[1, 2, 5] しかし、構造体および多次元配列アクセスのベクトル化は通常、非ユニットストライド方式の不規則なメモリーアクセスを制御するために、SIMD ハードウェアでサポートされている非ユニットストライド方式のロード / ストアとギャザー / スキャッター命令を使用します。

プログラマーが明示的なベクトル・プログラミングにより効率的なベクトル並列処理を行う方法は、構造体配列 (AOS) を配列構造体 (SOA) に変換し、配列次元のアクセス順を変更して、SIMD ベクトル化を適用することです。C/C++ 配列アクセスでは、SIMD を最内次元に適用します。Fortran 配列アクセスでは、SIMD を最外次元に適用します。目標は、ユニットストライド方式のメモリーアクセスを行えるように配列の形状を変更するか配列アクセスを変更することです。

BLOG HIGHLIGHTS

「The Future of Fortran」の Doctor Fortran

STEVE LIONEL ›

2014 年 11 月に、私は、SC14 (以前「Supercomputing」と呼ばれていたイベント) で「The Future of Fortran (Fortran の将来)」という名前のセッションを行いました。私は、このセッションに Fortran 標準委員会のほかのベンダーやメンバーの代表の一人として参加するように依頼を受けたと思っていたのですが、いざセッションが始まると、そこにいた関係者は私一人だったのです！まぁ、それはさておき。

私は、Fortran 標準規格の現状と、現在 Fortran 2015 (この名前になることを期待しています) と呼ばれている次の標準規格について、短い、ベンダー・ニュートラルの中立的なプレゼンテーションを準備していました。質疑応答は、予想よりもはるかに本格的でした (時間が短いにもかかわらず質問者は 70 ～ 80 名ほどいました)。セッションの最後に出席者の皆さんに参加を依頼した、Fortran の使用法に関するオンライン調査の結果は興味深いものでした。

この記事の続きはこちら (英語) でご覧になれます。›

明示的なベクトル・プログラミングの例

このセクションでは、C/C++ および Fortran の明示的なベクトル・プログラミングの例を紹介します。実際には、ベクトル化されたループが正しく実行されることをプログラマーが知っていても、ループが複雑な場合やループに潜在的な依存性がある場合、コンパイラーはループをベクトル化しないことがあります。高度なコンパイラーは、単純なケースに対してランタイムテストを行ったり複数のバージョンを生成できますが、コードサイズやコンパイル時間が増加します。SIMD 構文は、これらの問題の解決に役立ちます。アプリケーション・プログラマーは、SIMD 構文を使用して、ループがベクトル化できることをコンパイラーに保証することができます。

```
void star( double *a, double *b, double *c, int n, int *ioff)
{  #pragma omp simd
   for ( int i = 0; i < n; i++) a[i] *= b[i] * c[i+ *ioff];
}

subroutine star(a,b,c,n,ioff_ptr)
   implicit none
   double precision  :: a(:),b(:),c(:)
   integer           :: n, i
   integer, pointer  :: ioff_ptr
   !$omp simd
   do i = 1,n
         a(i) = a(i) * b(i) * c(i+ioff_ptr)
   end do
end subroutine
```

15 C++ と Fortran で記述した SIMD ループの例

図 15 の例 (OpenMP* C++ および Fortran の両方で記述) は、int 型のポインター *ioff がコンパイル時に不明であることを示しています。そのため、コンパイラーは、関数 / サブルーチン star を呼び出すたびに、*ioff が負の整数値または正の整数値のいずれかであると仮定しなければいけません。また、コンパイラーは a、b、c がエイリアスされるかどうか知りません。例えば、a と c がエイリアスされて *ioff = -2 の場合、このループには後方へのループ伝搬依存性が含まれるためベクトル化できません。しかし、プログラマーが *ioff は正の整数であると保証できれば、a と c がエイリアスされる場合でもループをベクトル化することができます。プログラマーは、SIMD 構文を使用してこの特性を保証することができるのです。

図 16 の 2 つ目の例 (OpenMP* C/C++ と Fortran の両方で記述) は、SIMD 構文を再帰関数に使用できることを示しています。関数 **fib** は、main では無条件で呼び出され、関数内では条件付きで再帰的に呼び出されています。デフォルトでは、コンパイラーは、オリジナルのスカラーバージョンを保持したまま、マスクされたベクトルバージョンとマスクされていないベクトルバージョンを作成します。

```
#include <stdio.h>
#include <stdlib.h>
#define N 45
int a[N], b[N], c[N];

#pragma omp declare simd inbranch
int fib( int n )
{ if (n <= 2)
return n;
   else {
         return fib(n-1) + fib(n-2);
   }
}
int main(void)
{
for (int i=0; i < N; i++) b[i] = i;

   #pragma omp simd
   for (int i=0; i < N; i++) {
         a[i] = fib(b[i]);
}
   printf("Done a[%d] = %d\n", N-1, a[N-1]);
   return 0;
}

program Fibonacci implicit
   none integer,parameter  :: N=45
   integer                 :: a(0:N-1), b(0:N-1)
   integer, external       :: fib
   do i = 0,N-1
         b(i) = i
   end do
```

16 declare SIMD と SIMD 構文を使用した SIMD プログラム

```
    !$omp
    simd do
    i=0,N-1
          a(i) = fib(b(i))
    end do
    write(*,*) “Done a(“, N-1, “) = “, a(N-1) !44 1134903168 end program

 recursive function fib(n) result(r)
 !$omp declare simd(fib) inbranch
    implicit none
    integer     :: n, r if (n <= 2) then
          r = n
    else
          r = fib(n-1) + fib(n-2)
    endif
 end function fib
```

16 declare SIMD と SIMD 構文を使用した SIMD プログラム (続き)

fib は main では無条件で呼び出されていますが、プログラマーは fib に対するループのトップレベルの呼び出しを手動でインライン展開して、inbranch 節を使用できます。別の可能性は、例で仮定しているように、最近のコンパイラーがこのインライン展開を自動的に行い、fib が常に条件付きで呼び出されることです。どちらの場合も、マスクされたベクトルバージョンのみ生成するようにコンパイラーに指示するため、コンパイル時間とコードサイズを減らすことができます。

ケーススタディー : SIMD とスレッド化のシームレスな統合

OpenMP* 4.0 は、最近のプロセッサーの能力を引き出すために、スレッドレベルの並列処理とベクトルレベルの並列処理の両方を利用できる効率的なモデルを提供します。例えば、インテル® Xeon Phi™ コプロセッサーは、スレッドレベルの並列処理とベクトルレベルの並列処理の両方が適切に統合された方法で利用する必要があります。並列化の詳細については本題から外れるためここでは詳しく取り上げませんが、SIMD ベクトル拡張は、OpenMP* コンパイラーでサポートされている合成 / 組み合わせ構文 (parallel for SIMD) を使用して、あるいは異なるループレベルで別々に使用して、OpenMP* 4.0 のスレッド化とシームレスに統合できることを覚えておいてください。

図 17 は、一連の複素数からマンデルブロ集合 (有名な 2D フラクタル図形) の部分集合を表すグラフィカル・イメージを計算する Mandelbrot の例を示しています。このプログラムは、集合の内側と外側の点の数を出力します。

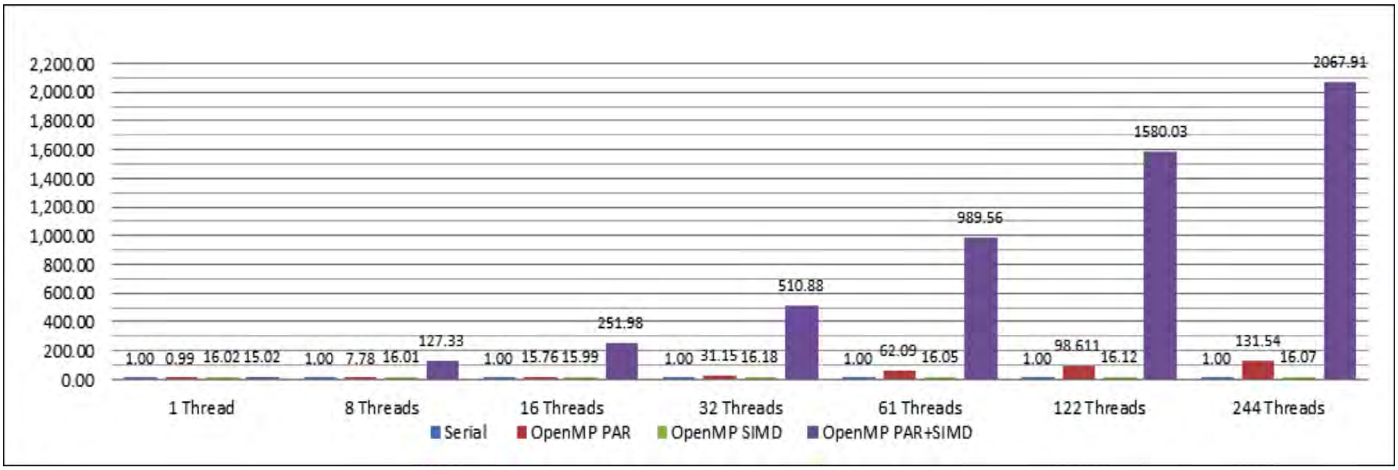
```
#pragma omp declare simd uniform(max_iter) simdlen(32)
uint32_t mandel(fcomplex c, uint32_t max_iter)
{
   // 外部マンデルブロ集合
   // uint32_t count = 1; fcomplex z = c;
   // のパラメーター c となる反復回数 (count 変数) を計算する

while ((cabsf(z) < 2.0f) && (count < max_iter)) {
         z = z * z + c;
         count++;
   }
   return count;
}
```

呼び出し元のコード:

```
int main() {
… … …
   #pragma omp parallel for schedule(guided)
   for (int32_t y = 0; y < ImageHeight; ++y) {
         float c_im = max_imag - y * imag_factor;
         #pragma omp simd safelen(32)
         for (int32_t x = 0; x < ImageWidth; ++x) {
         fcomplex in_val;
         in_val = (min_real + x*real_factor) + (c_im*1.0iF);
         count[y][x] = mandel(in_val, max_iter);
         }
   }
   … … …
}
```

17 OpenMP* parallel for と SIMD 構文を使用した例

関数 **mandel** は、ホットな関数で SIMD ベクトル化の候補であるため、**declare simd** で注釈を付けています。呼び出し元でホットなループは、二重の入れ子のループです。外部ループはスレッド化のために **parallel for** で注釈が付けられ、内部ループはベクトル化のために **simd** で注釈が付けられます (**図 17** を参照)。mandel の各呼び出しは while ループの終了条件により実行時間が異なる作業を実行するため、guided スケジューリング・タイプを使用してスレッド間のロードバランスを取っています。パフォーマンス測定は、インテル® Xeon Phi™ コプロセッサー 61 コア (1.00GHz、32KB L1、512KB L2、コアあたり 4 ハイパースレッド)、電力バジェット 300W、7936MB、16 メモリーチャネル、メモリー速度 2.75GHz、メモリー帯域幅 352GB/ 秒、512 ビット SIMD ベクトル長、メモリーデータ幅 32 ビットのシステムで行われました。



18 OpenMP* parallel for と SIMD を使用した Mandelbrot ワークロードのスピードアップ

図 18 は、SIMD ベクトル化のみ **(options -mmic -openmp -std=c99 -O3)** で、シリアル実行の約 16 倍のスピードアップを達成していることを示しています。また、OpenMP* 並列化により、**ハイパースレッディング** 無効 (61 コアで 61 スレッド) でシリアル実行の 62.09 倍のスピードアップ、ハイパースレッディング有効 (61 コアで 244 スレッド、コアあたり 4 スレッド) でシリアル実行の 131.54 倍のスピードアップを達成しています。並列化とベクトル化実行を組み合わせると、244 スレッドで実に 2067.91 倍のスピードアップを達成しています。1 スレッドから 61 スレッドまでのパフォーマンス・スケーリングは線形に近くなっています。ワークロードの計算リソースの競合が少なく、4 ハイパースレッディング・スレッドでレイテンシーが隠蔽されているため、ハイパースレッディングのサポートによるパフォーマンスの向上 (244 スレッドのスピードアップと 61 スレッドのスピードアップを比較) は約 2 倍と、一般的なハイパースレッディング・テクノロジーによるパフォーマンスの向上 (20 ～ 30 パーセント) よりも高くなっています。

パフォーマンス測定

OpenMP* C/C++ 言語での SIMD 拡張の有効性とインテル® C/C++ コンパイラー 16.0 (記事執筆時点ではベータ版) のコンパイラー・サポートを評価するため、さまざまなワークロードを用いてパフォーマンス測定が行われました。コンパイラーにより生成される SIMD コードは、アーキテクチャー固有のチューニングと、**`-O3 -Qopenmp-simd -QxSSE4.2/-QxCore-AVX2`** オプションを使用した積極的なメモリーの一義化による高度なスカラー、メモリー、ループの最適化により、高度に最適化されます。パフォーマンス測定は、インテル® Xeon® プロセッサー E3-1270 (開発コード名 Haswell) システム (4 コア、ハイパースレッディング有効)、動作周波数 3.50GHz、32GB RAM、8M L3 キャッシュ、64 ビット Windows Server* 2012 R2 上で行われました。この記事では、インテル® SSE4.2 とインテル® AVX2 のパフォーマンス結果を示します。

ワークロード

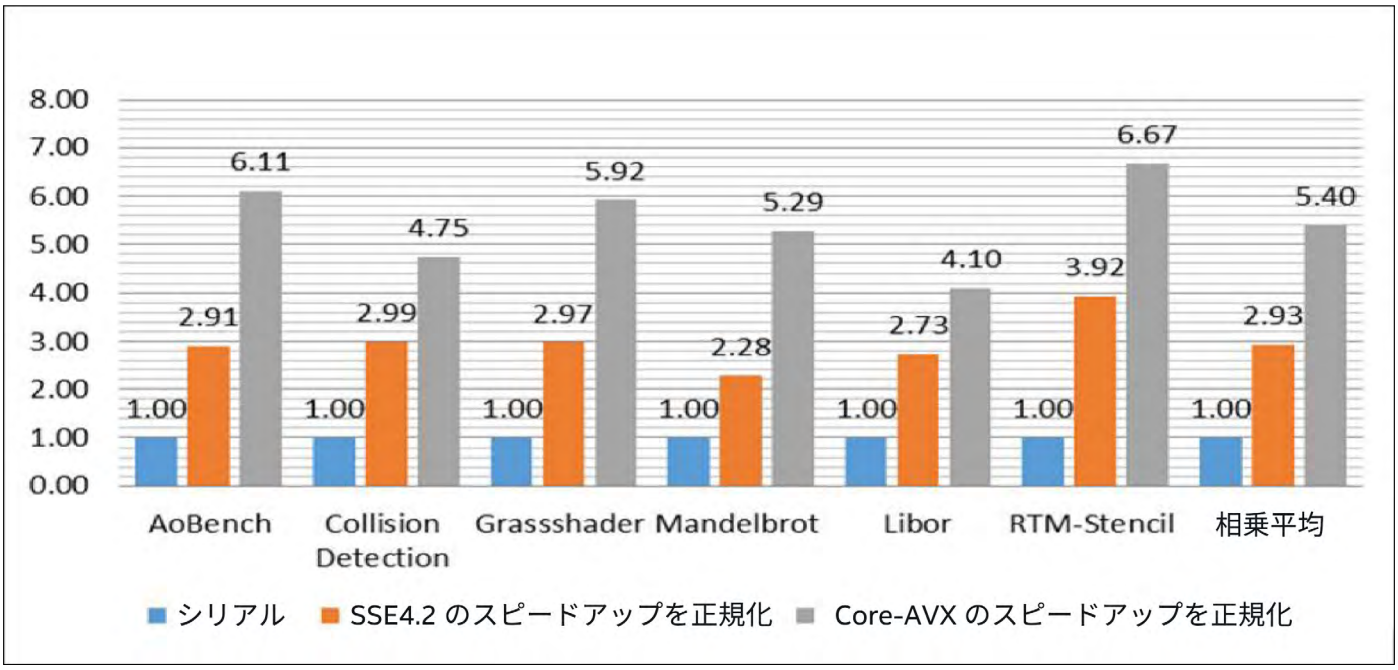
SIMD ベクトル拡張とコンパイラー実装のパフォーマンスを測定するため、異なるアプリケーション領域から 6 つのワークロードを選択しました。これらのワークロードは、ビジュアル・コンピューティング、グラフィック・シミュレーション、抽象的な数学計算、金融計算に加えて、アンビエント・オクルージョン (非反射面を作成することで、3D グラフィックの局部的反射モデルにリアルさを追加するシェーディング手法) のレンダリングに用いられる AOBench のような動的システム・シミュレーションもカバーしています。**表 1** は、使用される SIMD ベクトル拡張、入力データのサイズ、コードの行数、制御フローの特長、言語、各ワークロードの簡単な説明を含む、これらのワークロードの情報をまとめたものです。

ベンチマーク	ベンチマークの特性					
	使用されているベクトル拡張	データサイズ	コードの行数 (LOC)	制御フロー	言語	説明
AOBench	`pragma omp simd reduction; pragma omp declare simd uniform simdlem`	512x512	667	ベクトル関数の複数の分岐	C++	グラフィックス : (アンビエント・オクルージョン) シェーダーでオクルージョン (遮蔽) による減衰を計算
Collision Detection	`pragma omp simd; pragma omp declare simd uniform linear simdlen`	3125x32	330	ベクトル関数の単一の分岐	C++	グラフィックス : 一定間隔で衝突に基づいて動きを修正するために球体の移動のシーンを更新
Grassshader	`pragma omp simd reduction; pragma omp declare simd uniform linear`	1000x4x4	862	ベクトル関数の複数の分岐	C++	グラフィックス : 現実的な方法でのどかな自然の景色 (草、木、茂みなど) をシミュレート
Mandelbrot	`pragma omp simd; pragma omp declare simd uniform linear simdlen`	1000x1000	71	早期に終了するベクトル関数の for ループ	C	抽象的数学 : 複雑な 2 次再帰方程式の計算
Libor	`pragma omp simd uniform vectorlength`	80x40x15	400	ベクトル関数内部の for ループ	C	スワップション・ポートフォリオを計算
RTM Stencil	`pragma omp simd`	400x400x400 10 回	210	単純な入れ子の for ループ	C	地震 : リバース・タイム・マイグレーション、時間ステップの 3D ステンシルを計算 (FDTD)

表 1. ワークロード (ベンチマーク) の情報

パフォーマンス結果

パフォーマンスの向上率は、このセクションの最初に説明したインテル® アーキテクチャー・ベースのシステム上でシリアル実行と SIMD ベクトル実行の時間を測定して計算しました。**図 19** は、選択したワークロードの SIMD スピードアップを正規化したグラフです。これらのワークロードの SIMD スピードアップは 2.28 倍から 6.67 倍で、相乗平均は 2.93 倍 (インテル® SSE4.2 を使用) および 5.40 倍 (インテル® AVX2 を使用) でした。



19 インテル® SSE4.2 およびインテル® AVX2 を使用した複数のワークロードの SIMD スピードアップ

まとめ

最近の CPU や GPU プロセッサーにおける SIMD アーキテクチャーの普及に合わせて、[5, 6, 7] OpenMP* 4.0 (および 4.1 ドラフトバージョン) では、業界標準のハイレベルな SIMD ベクトル拡張のセットを利用できるように、[8] インテルの明示的な SIMD 拡張をサポートしました。[11] これらの拡張は、プログラマーとハードウェアの間に薄い抽象化レイヤーを形成します。プログラマーは、このレイヤーを使用して、生産性の低い SIMD 組込み関数やインライン・アセンブリー・コードを直接使用することなく、SIMD ベクトルユニットの計算能力を活用できます。これらの SIMD 拡張により、コンパイラー・ベンダーは最近の CPU や GPU プロセッサーのパフォーマンスを最大限に引き出すことができるようになりました。GPU アクセラレーターでは、インテルのオフロード機能と OpenMP* target 構文のサポートを組み合わせて 1024 ビット・ベクトル長をサポートすることにより、OpenMP* SIMD 拡張はインテル® GPU でシームレスに動作します。オフロードと GPU サポートの詳細およびパフォーマンス結果は別の号で紹介する予定です。

- インテル® C/**C++ コンパイラー**とインテル® **Fortran コンパイラー**は、通常の C/C++ および Fortran 関数をベクトル化するように拡張されました。従来のループ / ループの入れ子のみの SIMD **ベクトル化**を超えた、スレッドモデルとのシームレスな統合により、SIMD ベクトル・アーキテクチャー上で動作するマルチメディア、グラフィックス、ビジュアル・コンピューティング、ピクセル、組込みアプリケーションのパフォーマンスを向上します。
- プログラマーは、汎用言語拡張を利用して、C/C++ および Fortran で SIMD ベクトル並列処理を表現することができます。インテル® コンパイラーは、最近の SIMD ハードウェア向けの SIMD コードを生成できる生産的なプログラミング環境を提供します。生産性の低い、常に自動ベクトル化が失敗する SIMD 組込み関数やインライン・アセンブリー・コードを直接コード化する必要はもうありません。

謝辞

Jennifer Yu、Michael Rice、Clark Nelson、Peter Karam、Shin Lee、Stan Whitlock の各氏は、OpenMP* C/C++ および Fortran で SIMD 拡張をサポートするため、C/**C++ コンパイラー**と **Fortran コンパイラー**のフロントエンドの拡張に取り組んでくれました。U.S./Novo Vectorizer チーム、HPO チーム、ScalarOpt、PCG チームの皆さんは、インテル® Xeon® アーキテクチャーとインテル® Xeon Phi™ アーキテクチャー向けのハイパフォーマンス・オプティマイザーとコード・ジェネレーターの開発に貢献してくれました。Alice S. Chan、Kevin J. Smith、Geoff Lowney の各氏からは、管理面での支援をいただきました。最後に、ハイパフォーマンスなインテル® C/C++ コンパイラーとインテル® Fortran コンパイラーを開発していただいたインテル® コンパイラー・グループの皆さんに深く感謝します。

参考文献

1. A. Bik, M. Girkar, P. M. Grey, and X. Tian. "Automatic Intra-Register Vectorization for the Intel Architecture." International Journal of Parallel Programming, (2): pp. 65-98, April 2002.
2. A. Eichenberger, K. O'Brien, P. Wu, T. Chen, P. Oden, D. Prener, J. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind. Optimizing Compiler for the CELL Processor. In Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques, 2005.
3. D. Nuzman, I. Rosen, and A. Zaks. Auto-Vectorization of Interleaved Data for SIMD. In Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, 2006.
4. G. Ren, P. Wu, and D. Padua. A Preliminary Study on the Vectorization of Multimedia Applications for Multimedia Extensions. 16th International Workshop of Languages and Compilers for Parallel Computing, October 2003.
5. I. Buck, T. Foley, D. Horn, J. Superman, K. Patahalian, M. Hourston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware, ACM Transactions on Graphics, 23(3): pp. 777-786, 2004.
6. Intel Corporation. "Intel® Xeon Phi™ Coprocessor System Software Developers Guide," November 2012, http://software.intel.com/en-us/mic-developer.
7. Intel Corporation. Intel® Advanced Vector Extensions Programming Reference, Document Number 319433-011, June 2011.
8. OpenMP Architecture Review Board. "OpenMP Application Program Interface," Version 4.0, July 2013, and 4.1 (draft version), June 2015, http://www.openmp.org.
9. P. Wu, A. E. Eichenberger, and A. Wang. Efficient SIMD Code Generation for Runtime Alignment. In Proceedings of the Symposium on Code Generation and Optimization, 2005.
10. S. Larsen and S. Amarasinghe. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, pp.145-156, June 2000.
11. X. Tian, H. Saito, M. Girkar, S. Preis, S. Kozhukhov, A.G. Cherkasov, C. Nelson, N. Panchenko, and R. Geva. Compiling C/C++ SIMD Extensions for Function and Loop Vectorization on Multicore-SIMD Processors. In Proceedings of the IEEE 26th International Parallel and Distributed Processing Symposium – Multicore and GPU Programming Models, Language, and Compilers Workshop, pp. 2349–2358, 2012.
12. X. Tian, H. Saito, S. Kozhkhov, K.B. Smith, R. Geva, M. Girkar, and S. Preis, Vector Function Application Binary Interface, Version. 0.9.5, January 2013, https://www.cilkplus.org/sites/default/files/open_specifications/Intel-ABI-Vector-Function-2012-v0.9.5.pdf.

ベクトル化アドバイザー

ベクトル化を支援する新しいツール

Kevin O'Leary インテル コーポレーション テクニカル・コンサルティング・エンジニア
Kirill Rogozhin インテル コーポレーション テクニカル・コンサルティング・エンジニア

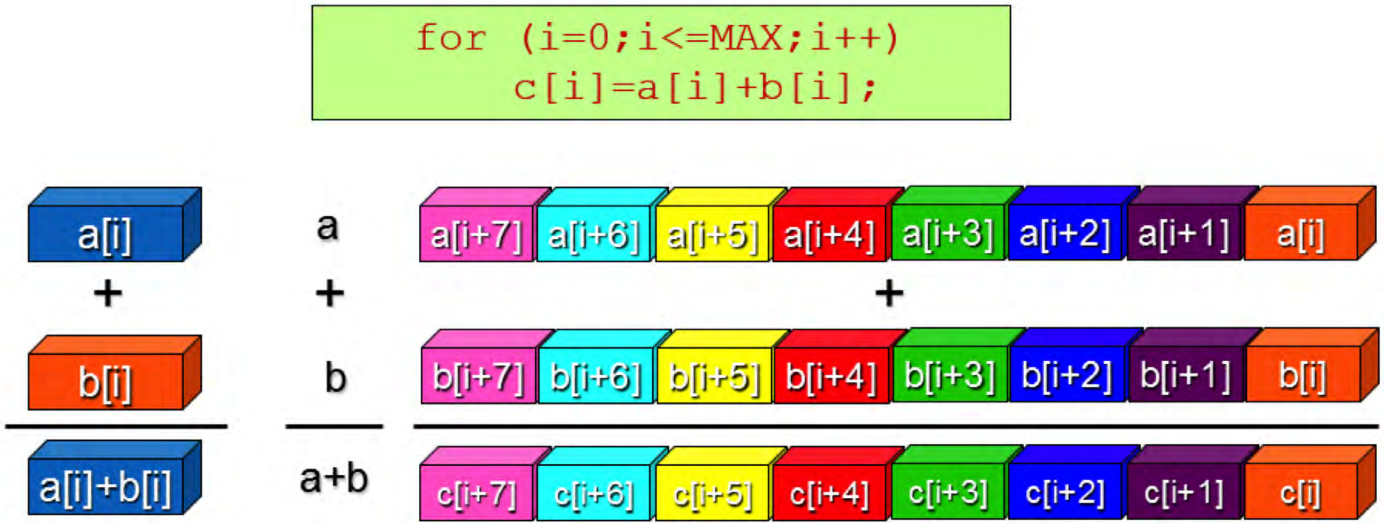
インテル® Advisor XE の最新リリースは、コードのベクトル化を支援します。**SIMD** 対応アプリケーションを深く解析して、次のような質問への答えを提供します。最も時間のかかるループはベクトル化されているか？ ベクトル化されていない場合、何が制約になっているのか？ ベクトル化されている場合、最適な命令セットを使用しているか？SIMD ループの効率の悪いピール / 剰余、均一でないメモリーアクセス、その他の問題のように、最適でないパターンはあるか？

インテル® Advisor XE は、ダイナミック解析、スタティック・バイナリー解析、コンパイラー・レポートを組み合わせます。利用可能なデータソースをほぼすべて活用して、ユーザーコードに関する最も包括的な情報 (CPU 時間、ループのトリップカウント、ベクトル長と命令セット、メモリー・アクセス・パターンなど) を提供します。インテル® Advisor XE はまた、詳細な統計を利用した、ボトルネックを解消するための推奨事項を示します。

この記事では、サンプルコードでインテル® Advisor XE を使用してベクトルコードを最適化する方法を説明します。

ベクトル化とは？

まず、**ベクトル化**の概要とベクトル化が困難な理由について説明します。ベクトル化とは何でしょうか？**図 1** は、ループがベクトル命令をどのように使用するか示しています。



1 **SIMD** 命令

ベクトル化はなぜそれほど重要なのでしょうか？ ベクトル命令はスカラー命令よりも大幅に高速化できます。上記の例では、多くの異なるスカラー加算命令を実行する必要がありますが、ベクトル加算命令にすれば 1 つで済みます。この大幅な高速化を達成するには、ベクトル化を効率的に行っていることを確認する必要があります。

ベクトル化を効率的に行うには、**コンパイラー**がループをベクトル化するときに何を行っているか理解すると参考になります。

一般的なベクトル化されたループは次の 3 つの部分で構成されます。

- メインのベクトル本体 – 3 つの部分の中で最も高速です。ここでできるだけ時間を費やすことがポイントです。
- オプションのピール部分 – ループのアラインされていない参照に使用されます。スカラーか、より遅いベクトルを使用します。一般に、ループのピール部分を削除するには、メモリーがどのようにアラインされているか特定する必要があります。
- 剰余部分 – 反復回数 (トリップカウント) がベクトル長で分割できない場合に生成されます。スカラーか、より遅いベクトルを使用します。剰余ループを削除するには、ループのトリップカウントがベクトル長で割り切れるようにして、さらに、ベクトル化するのに十分な反復回数であることを確認します。

大きなベクトルレジスターでは、ループのピール / 剰余部分の反復回数はより多くなります。最新のハードウェアではベクトルレジスターがより大きくなっているため、この種の問題は今後さらに重要になるでしょう。インテル® AVX2 には 256 バイトのレジスターがあり、8 つの 32 ビット値、4 つの 64 ビット値などを格納できます。

ではここで、何が**ベクトル化**を困難にしているか考えてみましょう。

ベクトル化の妨げとなる、あるいは効率的にベクトル化できない、潜在的なベクトル化問題のリストを次に示します。注 : ベクトル化は複雑であるため、このリストは完全ではありません。

- 連続していないメモリーアクセス — ベクトル化は可能ですが効率的ではありません。
  - 配列への非ユニット・ストライド・アクセス :

    ```
    for (i=0; i < N; i+=2) // i を 2 でインクリメントすることはユニットストライドではありません。
    ```
  - ループ内の間接参照 :

    ```
    for (i=0; i < N; i++)
    A[B[i]] = C[i] * D[i] // A のメモリー参照を見つけるには B[i] を解釈する必要があります。
    ```
- ループ内の関数の呼び出し — ベクトル化は可能ですが、シリアル化されボトルネックの原因になります。この場合、SIMD 対応関数を作成すると大幅にパフォーマンスが向上します。
- データ依存性 :

  ```
  for (i=0; i < N; i++)
  A[i] = A[i-1] * C[i]    // ループの以前の反復を参照しています。
  ```
- 非常に大きなピールまたは剰余部分。
- ハードウェアで利用可能なインテル® アーキテクチャーよりも古いアーキテクチャーを使用する (例えば、CPU がインテル® AVX2 をサポートしている場合にインテル® SSE4 用のコードを生成する)。

インテル® Advisor XE は、既存の SIMD の非効率性と解決方法について詳細な情報を提供します。

BLOG HIGHLIGHTS

インテル® Parallel Studio XE 2016: HPC アプリケーションおよびビッグデータ解析用ツールスイート

JAMES REINDERS >

インテル® Parallel Studio XE 2016 (2015 年 8 月 25 日に提供開始) は、ハイパフォーマンス・コンピューティング (HPC) および技術計算アプリケーション用の最新の開発者キットです。コンパイラー、ライブラリー、デバッグ機能、解析ツールを含むこのスイートは、インテル® アーキテクチャーを対象としており、最新のインテル® Xeon® プロセッサー (開発コード名 Skylake) とインテル® Xeon Phi™ プロセッサー (開発コード名 Knights Landing) をサポートしています。インテル® Parallel Studio XE 2016 は、Fortran、C/C++、Java* コードを設計、ビルド、検証、チューニングするソフトウェア開発者を支援します。

この記事の続きはこちら (英語) でご覧になれます。 >

インテル® Advisor XE

インテル® Advisor XE は、次の手順によりコードの効率的なベクトル化を支援します。

- 調査 – コードが時間を費やしているループと詳細な SIMD 統計を表示します。
- トリップカウント – 各ループの反復回数と呼び出し回数を測定します。
- 推奨事項 – 問題の解決方法について具体的なアドバイスを示します。
- 依存性解析 – ダイナミックな依存性解析を行い、ループに反復間の依存関係がないか確認します。
- メモリー・アクセス・パターン解析 – ベクトル化に適した方法でメモリーにアクセスしているか確認します。

調査

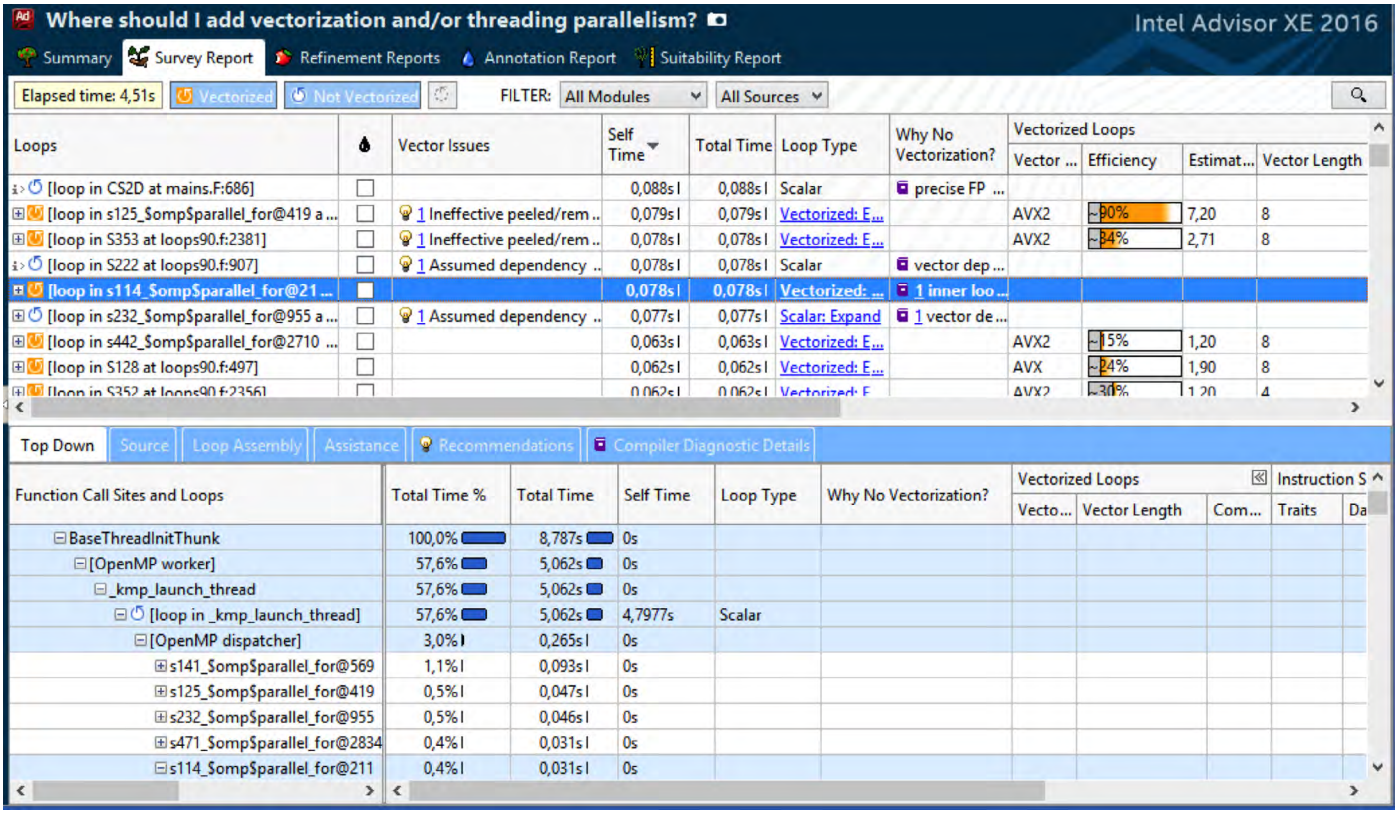
ベクトル化を追加してパフォーマンスを向上する最初のステップは、アプリケーションの調査です。この調査ステップで、時間を費やしているループが分かります。「ホットな」ループは、最適化による恩恵が最も得られる場所です。**図 2** は、アプリケーションの調査レポートです。インテル® Advisor XE では、ループの種類 (**ベクトル化**されて

いるかどうか) でフィルターすることができます。ベクトル化されていないループには、ループのベクトル化を妨げている原因について注釈が表示されます。ベクトル化されたループには、ベクトル化の効率に影響を与える可能性がある追加の詳細が表示されます。ベクトル化されたループには次の情報が表示されます。

- 効率と推定ゲイン – ループをベクトル化することにより達成されるスピードアップは？データサイズとベクトル長を計算することで、理論的に可能なスピードアップが分かります。例えば、ベクトル幅 256 ビットの インテル® AVX2 でベクトル要素として 32 ビット整数を使用している場合、最大のスピードアップは 256/32 = 8 倍になります。スピードアップが 8 倍に近ければ、ループはほぼ最適で、効率はほぼ 100% と言えるでしょう。
- ベクトル命令セット – インテル® SSE、インテル® AVX2、その他。より古い命令セットを使用することにより引き出せていないパフォーマンスがないか？
- データ型と特性 – 遅くなる可能性のある種類の命令 (抽出や挿入など) が使用されているか？
- ベクトル長とベクトル幅 – ベクトルをすべて利用しているか、一部のみ利用しているか？

さまざまな情報にすばやくアクセスできるように、調査レポートには次のタブが表示されます。

- Top down (トップダウン) – 選択したループとプログラムの残りの部分の関係を表示します。また、選択したループが内部ループなのか外部ループなのか簡単に確認できます。注 : 一般に、内部ループのみ自動的にベクトル化されます。
- Source (ソース) – プログラムのソースをスキャンして、コードにインライン展開されたコンパイラー・レポートを表示します。
- Loop Assembly (ループ・アセンブリー) – 生成された命令と実行に費やされた時間を表示します。
- Recommendations (推奨事項) – インテル® Advisor XE が推奨する問題の解決方法を表示します。
- **Compiler** Diagnostic Details (コンパイラー診断詳細) – 問題の解決に役立つ追加の詳細とサンプルコードを表示します。



2 調査レポート

トリップカウント

タイミング情報に加えて、ループの反復回数と呼び出されている回数を知ることも重要です (**図 3**)。インテル® Advisor XE は、別の解析を実行してこの情報を提供します。この情報を収集するプロセスはアプリケーションのオーバーヘッドを増やすため、トリップカウントは別の手順で収集されます。

Where should I add vectorization and/or threading parallelism? Intel Advisor XE 2016

Summary | Survey Report | Refinement Reports | Annotation Report | Suitability Report

Elapsed time: 14.82s | Vectorized | Not Vectorized | FILTER: All Modules | All Sources

Loops		Vector Issues	Self Time	Total Time	Trip Counts					Loop Typ
					Median	Min	Max	Call Count	Iteration Duration	
[loop at Multiply.c:53 in matvec]		1 Assumed de...	13.873s	13.873s	101; 3	101; 3	101; 3	12000000; 1...	< 0.0001s	Vectoriz
[loop at Multiply.c:53 in matvec]			0.030s	0.030s	3	3	3	1000000	< 0.0001s	Vectorize
[loop at Multiply.c:53 in matvec]		1 Assumed dep..	13.843s	13.843s	101	101	101	12000000	< 0.0001s	Vectorize
[loop at Multiply.c:45 in matvec]		1 Ineffective pe..	0.141s	14.329s	12; 2	12; 2	12; 2	1000000; 100...	< 0.0001s	Vectorize
[loop at Multiply.c:45 in matvec]			0.035s	0.380s	2	2	2	1000000	< 0.0001s	Remainde
[loop at Multiply.c:45 in matvec]			0.106s	13.949s	12	12	12	1000000	< 0.0001s	Vectorize
[loop at Multiply.c:53 in matvec]			0.345s	0.345s	101	101	101	2000000	< 0.0001s	Scalar
[loop at Driver.c:146 in main]		1 Assumed dep..	0.010s	14.408s	1000000	1000000	1000000	1	< 0.0001s	Scalar

3 トリップカウント

推奨事項

インテル® Advisor XE は、コードのベクトル化を改善すべき場所について推奨事項も示します。電球アイコンをクリックすると、コードの改善方法について詳細なフィードバックが表示されます (**図 4**)。

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

1 Issue: Ineffective peeled/remainder loop(s) present

2 All or some source loop iterations are not executing in the loop body. Improve performance by moving source loop iterations from peeled/remainder loops to the loop body.

Add data padding

The trip count is not a multiple of vector length. To fix: Do one of the following:

- Increase size of objects and add iterations so the trip count is a multiple of vector length.
- Increase the size of static and automatic objects, and use a compiler option to add data padding.

Windows* OS	Linux* OS
/Qopt-assume-safe-padding	-qopt-assume-safe-padding

Note: These compiler options apply only to Intel® Many Integrated Core Architecture (Intel® MIC Architecture). Option `-qopt-assume-safe-padding` is the replacement compiler option for `-opt-assume-safe-padding` , which is deprecated.

4 推奨事項

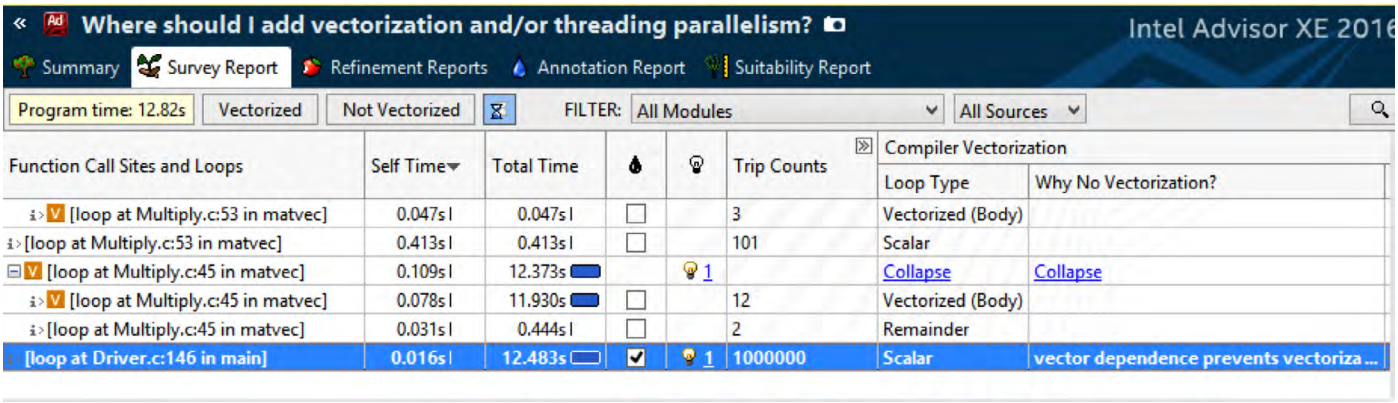
依存性解析

正しいコードを生成するため、コンパイラーは、コンパイルしている言語のセマンティクスに対して保守的な見地に立たなければいけません。言語の規則に基づいて依存性が存在する可能性がある場合は、依存性が存在すると仮定します。インテル® Advisor XE のような動的なツールを使用することにより、仮定した依存性が事実かどうか確認することができます。

2 つの結果が起こりえます。

- 実際の依存性はありません。この場合、**`#pragma omp simd`** や **`#pragma ivdep`** などの宣言子の使用に関してコンパイラーにヒントを与えます。
- 依存性が見つかりました。この場合、ループをベクトル化する前に依存性を除去する必要があります。ローカル変数を使用するか、ループやデータ構造などを変更します。

調査レポートで確認するループを選択して、依存性解析を実行します (**図 5**)。



5 ループの確認

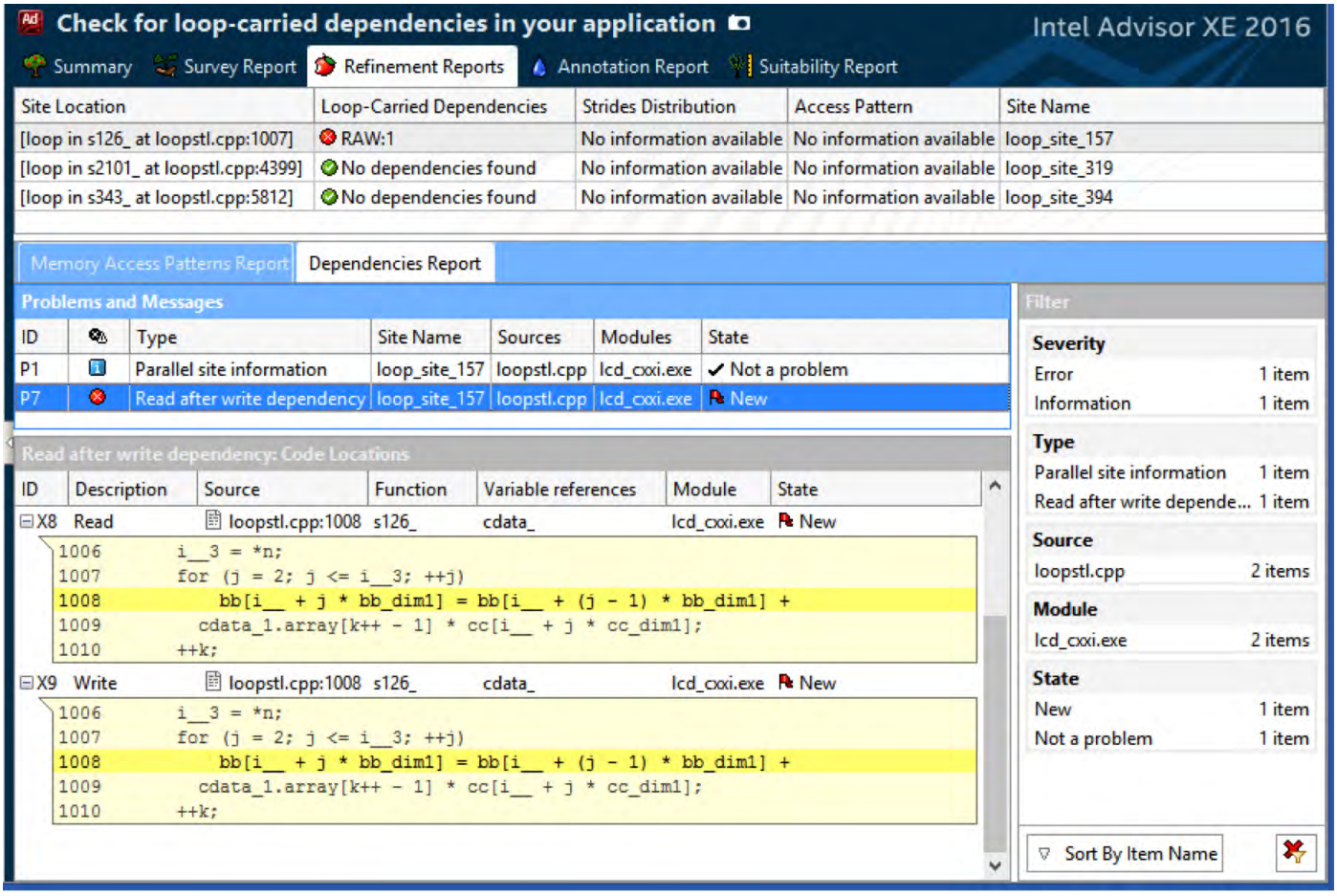
COSMOS チームが宇宙論コードで 100 倍のスピードアップを達成

The Parallel Universe では、ほかの出版物の関連記事を取り上げることがあります。この記事もその 1 つです。

HPCwire のこの記事は、インテル® Xeon Phi™ コプロセッサー (開発コード名 Knights Corner) 向けにコードを移植する作業で行われた最適化により 100 倍以上のスピードアップを達成した、ケンブリッジ大学の Stephen Hawking 氏の研究グループの成果を紹介しています。理論物理学者達は、MODAL と呼ばれるシミュレーション・コードを使用して、宇宙背景放射 (ビッグバンの名残であるマイクロ波背景放射) を調査しました。

この記事の続きはこちら (英語) でご覧になれます。 >

以下の依存性レポートは、4399 行と 5812 行のループに依存性がないことを示しています。そのため、これらのベクトル化を強制するコンパイラー・プラグマを使用しても安全です。1007 行のループには依存性があります。インテル® Advisor XE は、正確なソース行と依存性タイプ (書き込みの後の読み取り) を検出しています (**図 6**)。



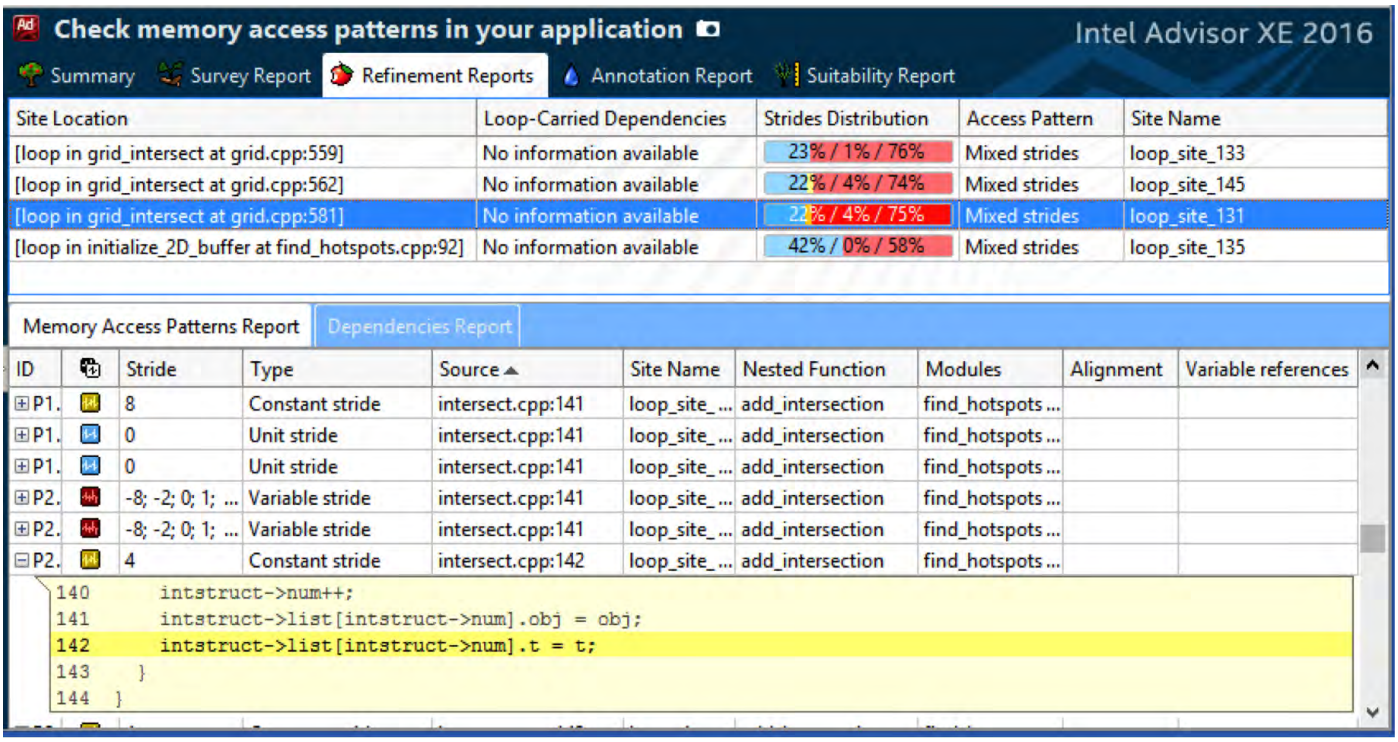
6 依存性レポート

メモリー・アクセス・パターン (MAP) 解析

データ構造がメモリー上にどのようにレイアウトされ、ループでどのようにアクセスされているか知っていれば、ベクトル化の効率を大幅に向上させることができます。メモリー参照がユニットストライド方式などで適切にアライメントされていることは非常に重要です。ベクトル化を支援する、メモリーアクセスに関連するいくつかの手法があります。構造体配列から配列構造体にデータ構造を変換すると効率的です。MAP 解析を使用すると、本質的にベクトル化が非効率なパターンを見つけ出すことができます。

インテル® Advisor XE の MAP 解析 (**図 7**) は、メモリーアクセス命令のソースコードとアセンブリー行、データ型とサイズ、アライメントなどを示します。次の 3 種類のメモリーアクセスを識別します。

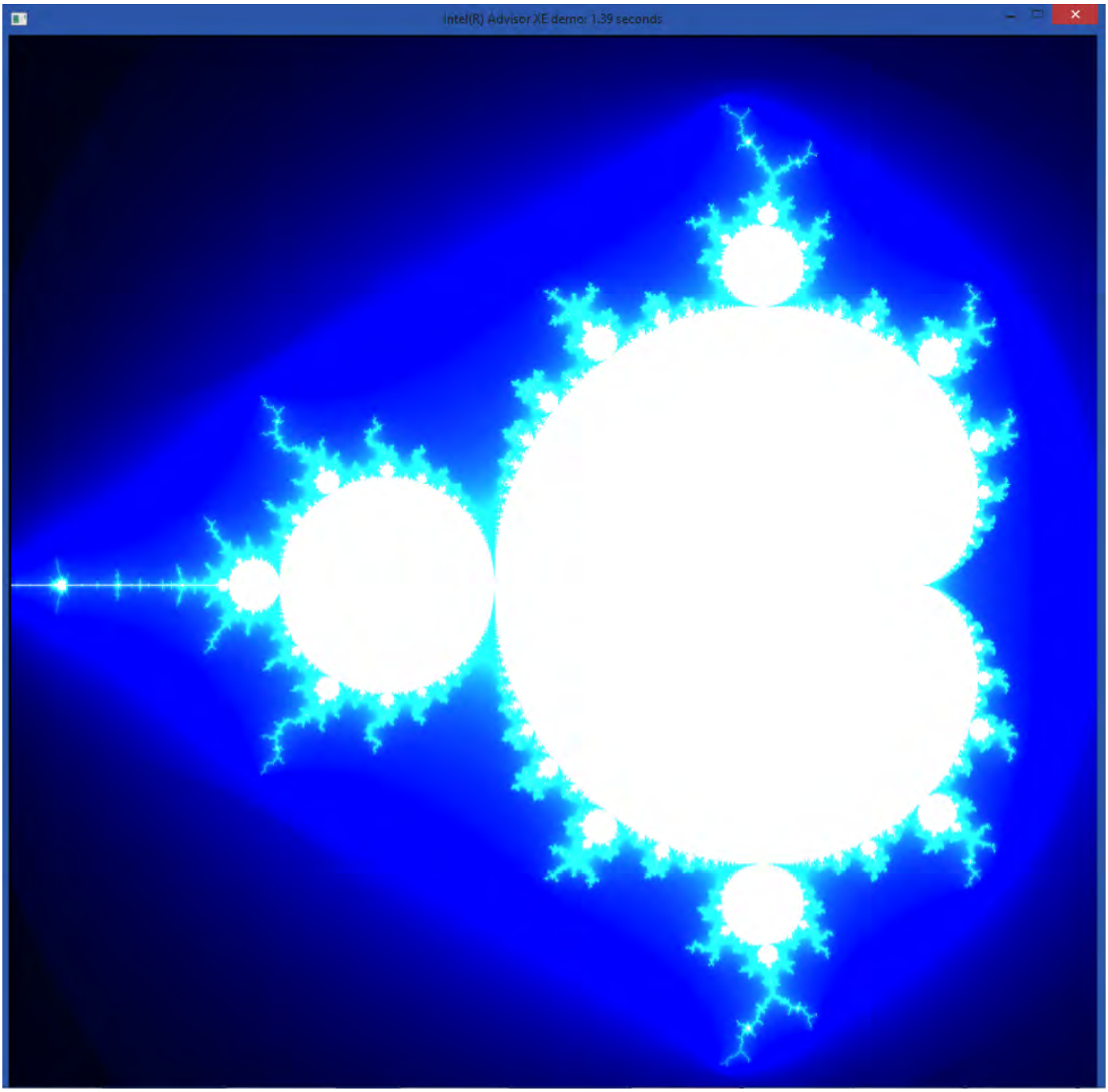
- ユニット・ストライド・アクセス – メモリーは読み取りまたは書き込みです。
- 定数ストライド – ストライドは 2 つ以上のデータ要素 (構造体配列など) です。
- 変数ストライド – パフォーマンスは最低です (ギャザー / スキャッター・パターン、配列インデックスなど)。



7 メモリー・アクセス・パターン解析レポート

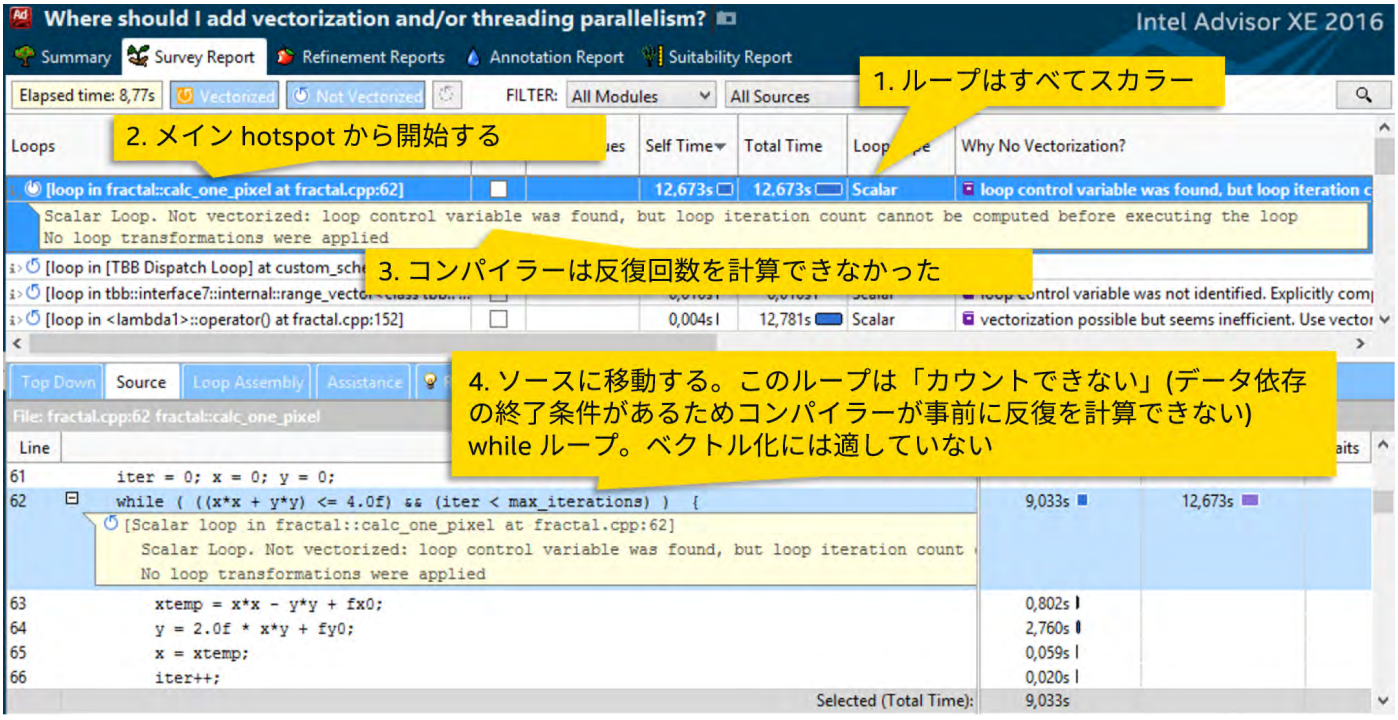
実サンプルの最適化フロー

ここでは、マンデルブロ・フラクタルを計算して表示するサンプル・アプリケーション (**図 8**) を使用して、インテル® Advisor XE のベクトル化アドバイザーのフローを説明します。サンプルはインテル® TBB ライブラリーを使用してすでに並列化され、CPU コアを効率的に利用しています。サンプルが SIMD 命令を活用しているか確認しましょう。



8 Mandelbrot サンプル・アプリケーション・ウィンドウ

最初のステップは、インテル® Advisor XE の調査解析 (**図 9**) を実行することです。[Loop Type] 列から、アプリケーションのすべてのループがスカラーであることが分かります。最も CPU 時間を費やしているループは、fractal.cpp の 62 行目のスカラーループです。右端の [Why No Vectorization?] 列には、コンパイラーがループの反復回数を計算できなかったと表示されています。



9 Mandelbrot のベースライン調査

BLOG HIGHLIGHTS

プロセッサーの音節分け — 一般的な言語による分解

JAMES REINDERS >

processor、coprocessor、microprocessor、multiprocessing はどのように発音しますか？

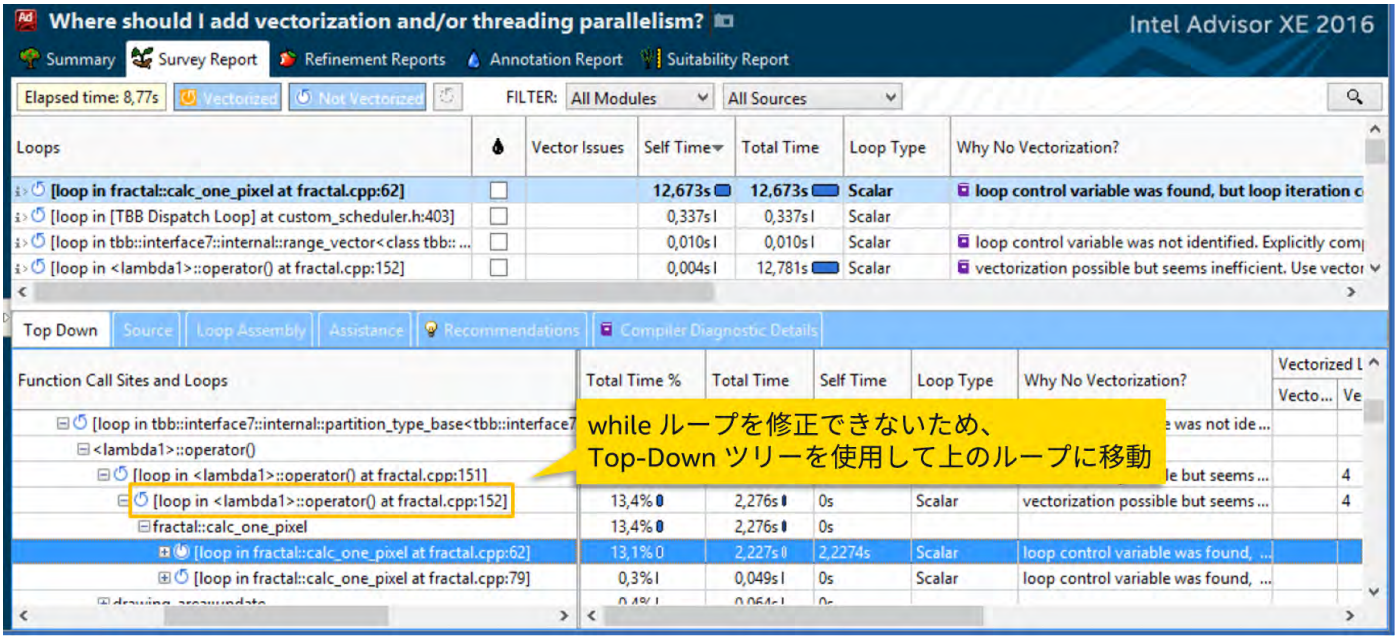
私が正しいと思っているように、proc-es-sor、co-proc-es-sor、mi-cro-proc-es-sor、mul-ti-proc-ess-ing のように区切って発音しますか？

それとも、pro-ces-sor、co-pro-ces-sor、mi-cro-pro-ces-sor、mul-ti-pro-cess-ing のように区切って発音しますか？

答えは、アメリカ英語を話しているか、イギリス英語を話しているかによって異なるでしょう。

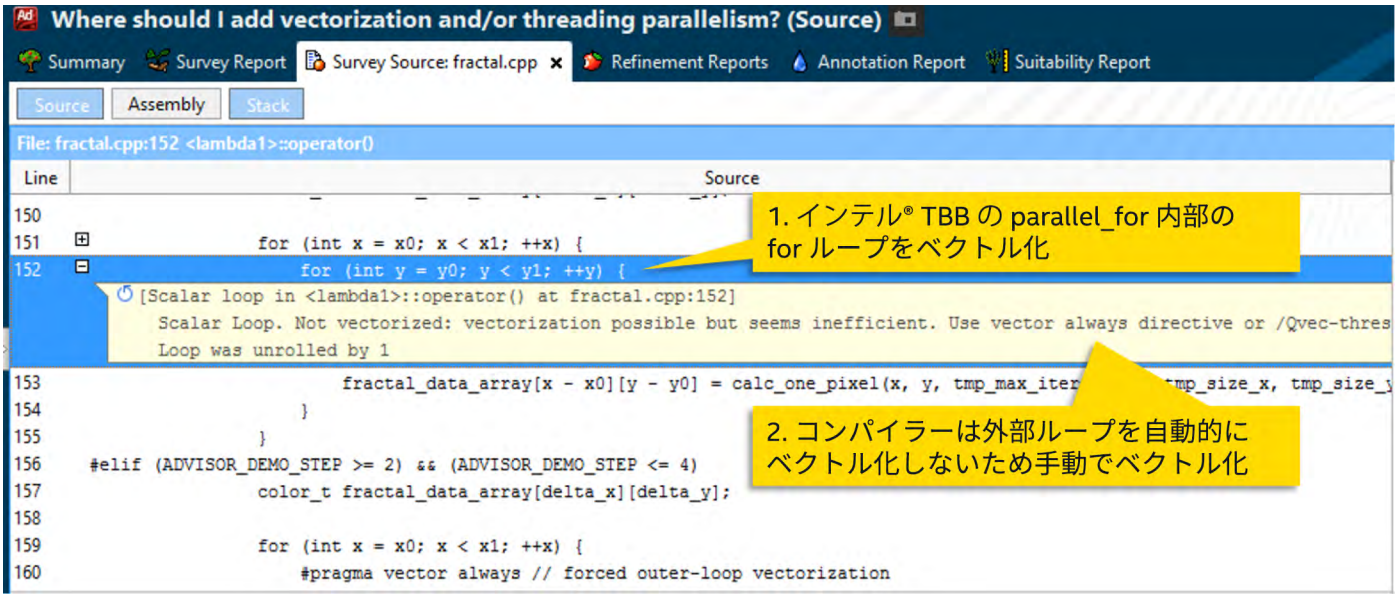
この記事の続きはこちら (英語) でご覧になれます。 >

[Source] タブを見ると、このループはデータ依存の終了条件を含む while ループであることが分かります。反復回数がコンパイル時に計算できなかった理由は明白です。この問題は簡単に修正できないため、代わりにほかの可能性を探すことにしましょう。[Top Down] 列に切り替えて、呼び出しツリーを調べます (**図 10**)。while ループは、152 行目で別のループによって呼び出された **`calc_one_pixel`** 関数から呼び出されています。



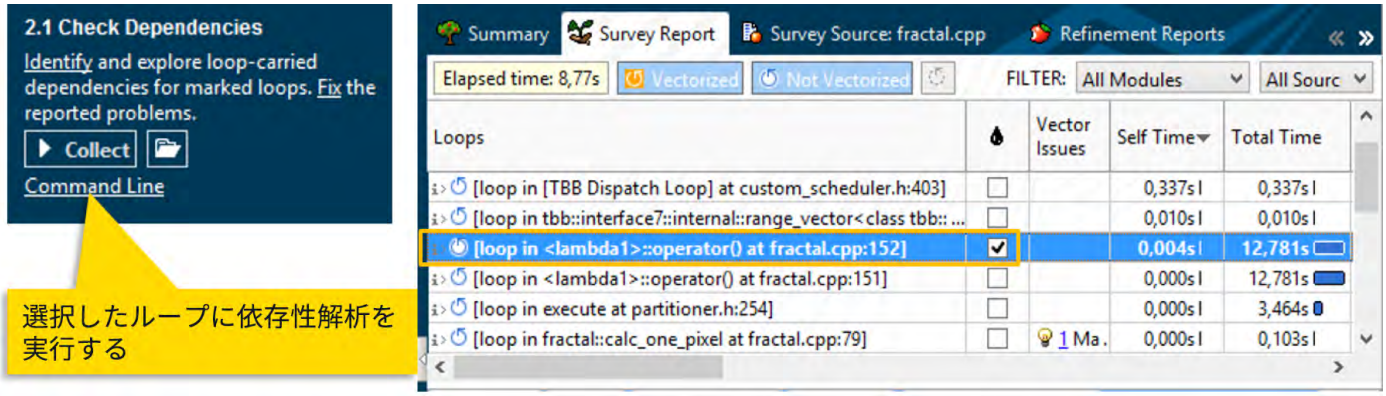
10 Mandelbrot の Top-Down ツリー

この外部ループをベクトル化できないか確認します。ループをダブルクリックしてソースビューを開きます (**図 11**)。

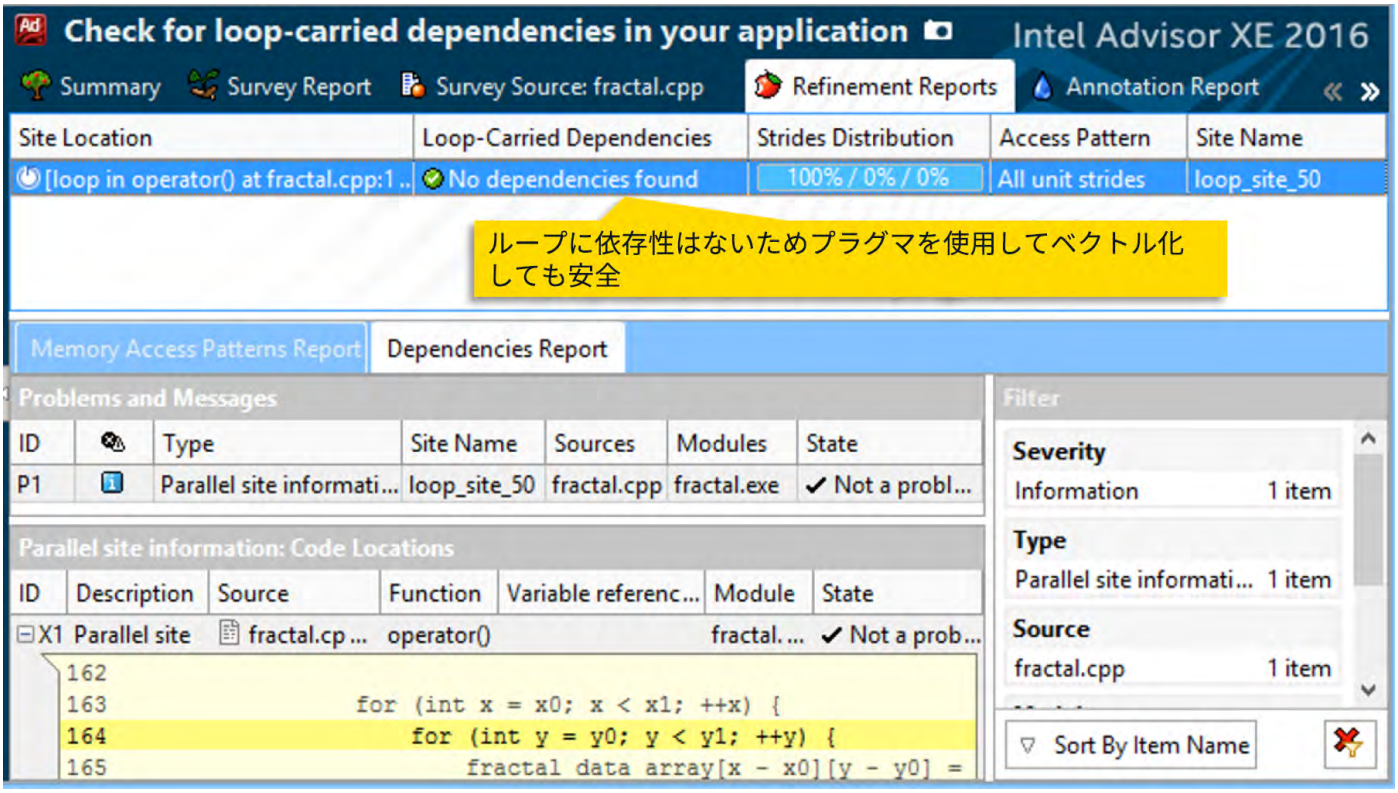


11 Mandelbrot の外部ループのソース

152 行目の外部ループは、インテル® TBB の **parallel_for** ループ本体から呼び出されている通常の for ループです。ソースの診断メッセージは、コンパイラーがこのループのベクトル化は効率的と見なさなかったことを示しています。これは通常、ほかの制約がなく、ベクトル化を手動で行うことができることを意味します。しかし、念のために、ループに反復間の依存関係がないか確認します (**図 12**)。



12 依存性の確認



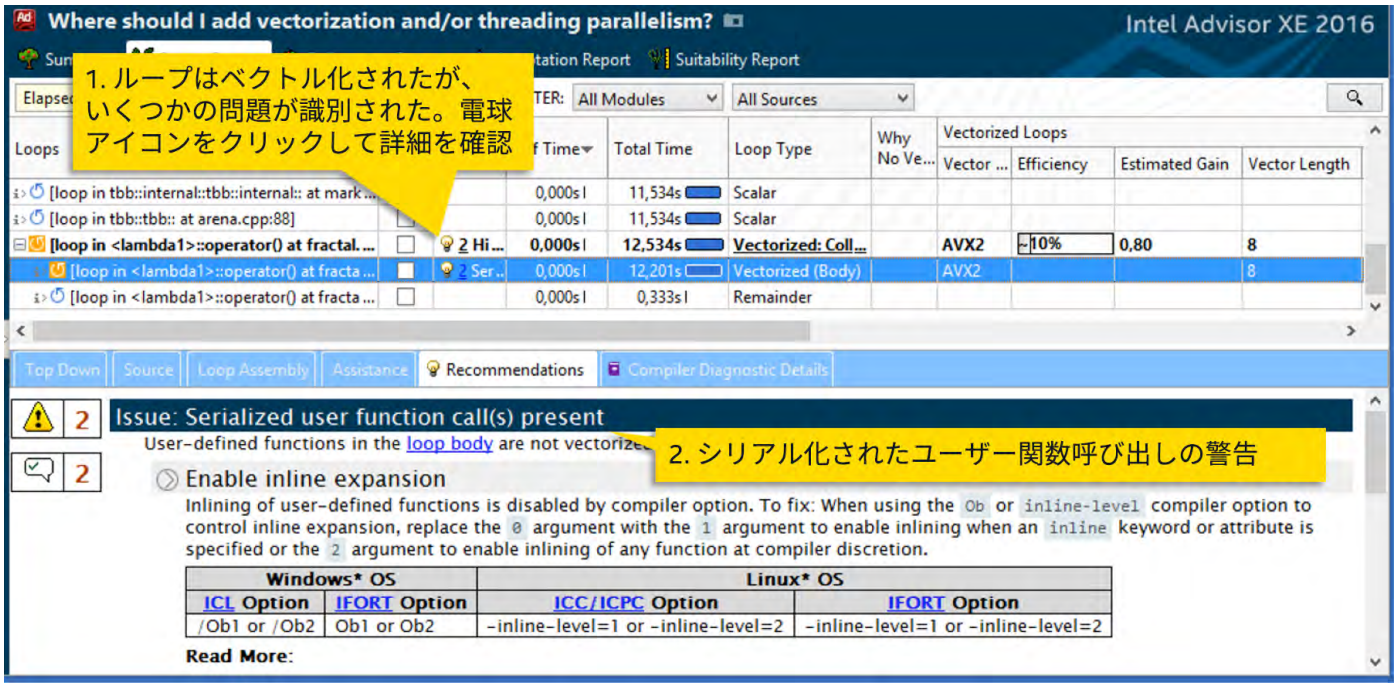
13 Mandelbrot の依存性レポート

依存性解析により、ループの反復間に依存性がないことが分かりました (**図 13**)。これで、例えば **#pragma vector always** 構文を使用して、このループを安全にベクトル化できます (**図 14**)。

```
color_t fractal_data_array[delta_x][delta_y];

for (int x = x0; x < x1; ++x) {
    #pragma vector always // forced outer-loop vectorization
    for (int y = y0; y < y1; ++y) {
        fractal_data_array[x - x0][y - y0] = calc_one_pixel(x, y, \
            tmp_max_iterations, tmp_size_x, tmp_size_y, tmp_magn, tmp_cx, tmp_cy, 255);
    }
}
```

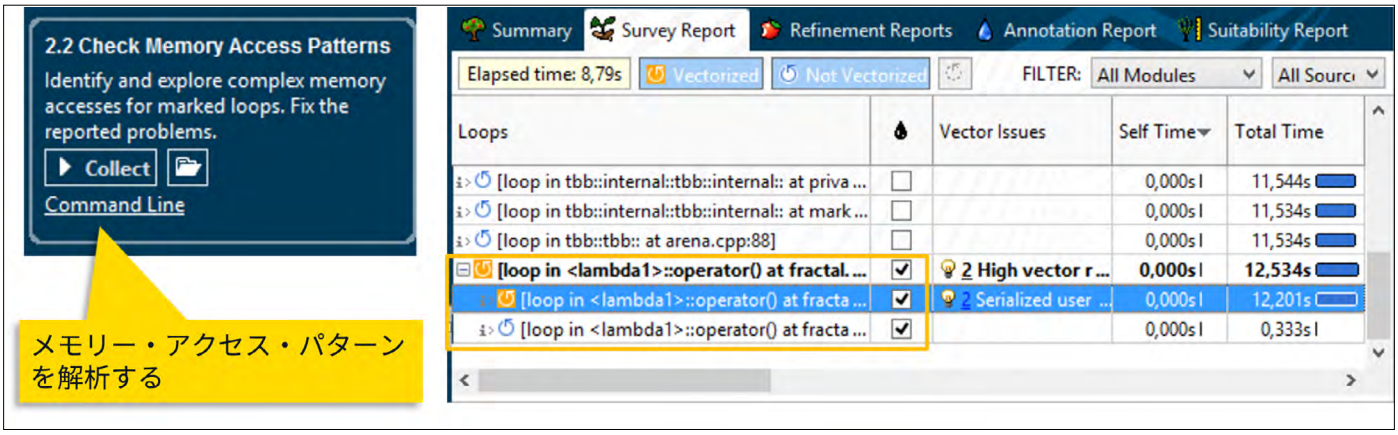
14 外部ループをベクトル化

ソースの変更後、ループはベクトル化されました。しかし、CPU 時間は改善されておらず、効率はわずか 10% で、推定ゲインは 0.8 です。ベクトル長は 8 (256 ビットのインテル® AVX2 レジスターで 32 ビット・データ型を処理) であるため、理想的なパフォーマンス・ゲインは 8 倍です。実際の数値は約 0.8 と、約 10 パーセントに過ぎず、スカラーバージョンのベースライン (1.0) を下回っています。SIMD ループはなぜ効率的でないのでしょうか？[Vector Issues] 列を調べてみましょう。いくつかの問題が電球アイコン付きで表示されています。アイコンをクリックして下部の [Recommendations] タブを確認します (**図 15**)。

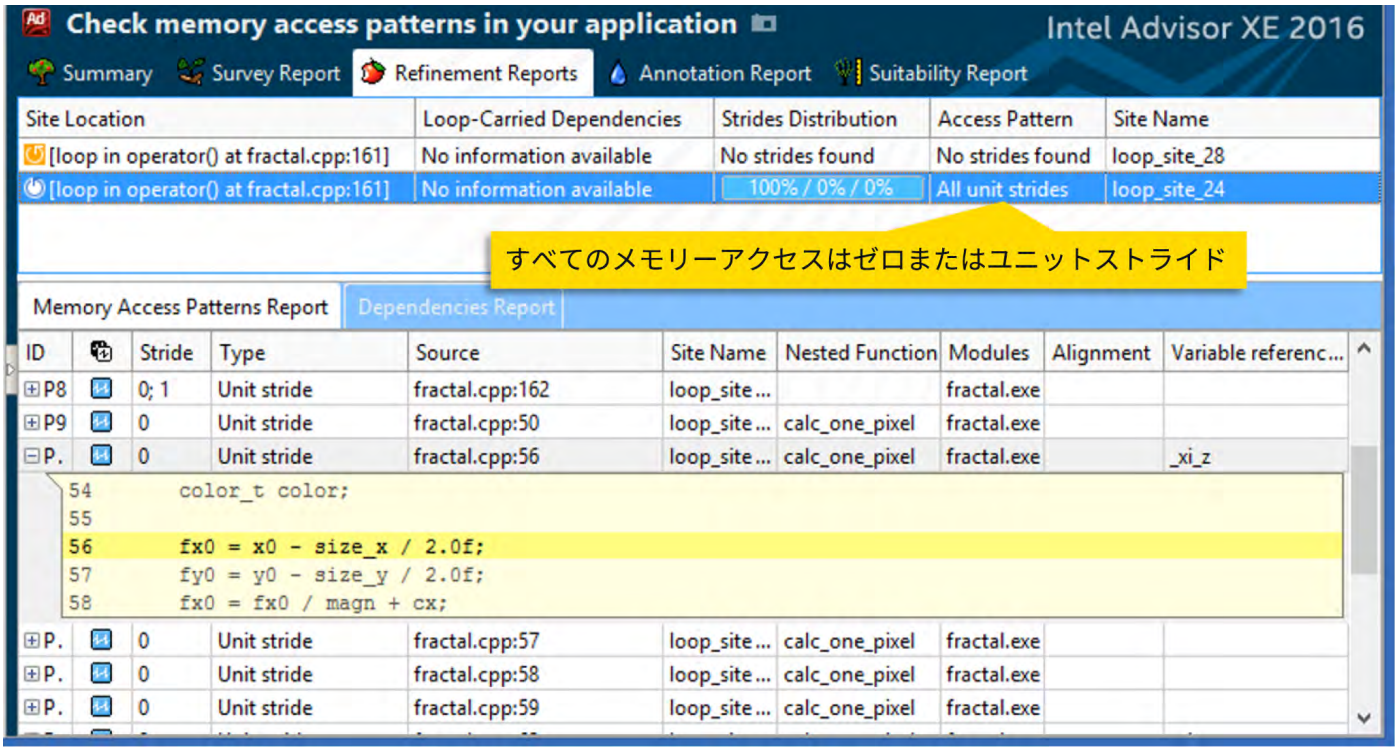


15 シリアル化された関数呼び出しに関する推奨事項

最初のヒントは、ループにシリアル化された関数呼び出しが含まれていることを示しています。これらの関数はスカラーデータを処理して SIMD ループの実行をシリアル化しているため、ボトルネックの原因になります。**図 15** のコード部分を調べると、ループ内の関数は **`calc_one_pixel`** のみであることが分かります。この号の前の記事で説明したように、OpenMP* 4.0 SIMD 構文を使用して、該当部分を手動でベクトル対応にすることができます。しかし、そのためには、関数パラメーターにメモリーアクセスの種類 (uniform、vector、または linear) を指定する必要があります。実際のアクセスパターンを決定するため、該当ループを選択して、メモリー・アクセス・パターン解析タイプを実行します (**図 16**)。呼び出される関数も解析されます。



16 メモリーアクセスの確認



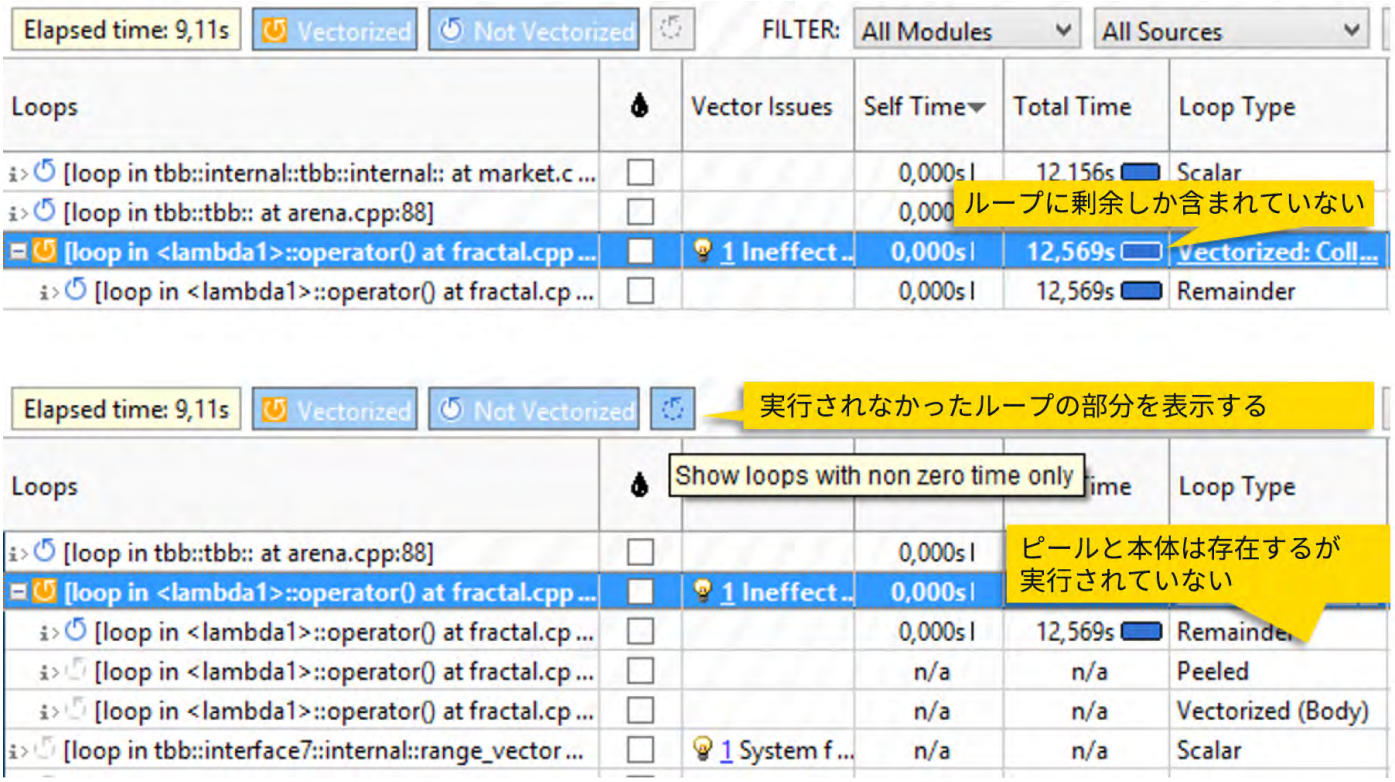
17 Mandelbrot の MAP レポート

解析結果は、**calc_one_pixel** 関数のすべてのメモリーアクセスがユニットストライドであることを示しています (**図 17**)。ほとんどのアクセスはストライド 0、つまり各ループ反復で同じデータを読み取りまたは書き込みしています。1 つのアクセスのみストライド 1 で、ループ・インデックス変数 y と関係があるようです。変数 y (アクセスタイプ「linear」) を除くすべてのパラメーターのアクセスタイプを「uniform」で宣言して関数をベクトル対応にします。インテル® Advisor XE の調査レポートに従ってベクトル長は 8 にします (**図 18**)。

```
#pragma omp declare simd uniform(x0, max_iterations, size_x, size_y, magn, cx, cy, gpu) \
    linear(y0:1) simdlen(8)
color_t fractal::calc_one_pixel(int x0, int y0, int max_iterations, int size_x, \
    int size_y, float magn, float cx, float cy, int gpu)
```

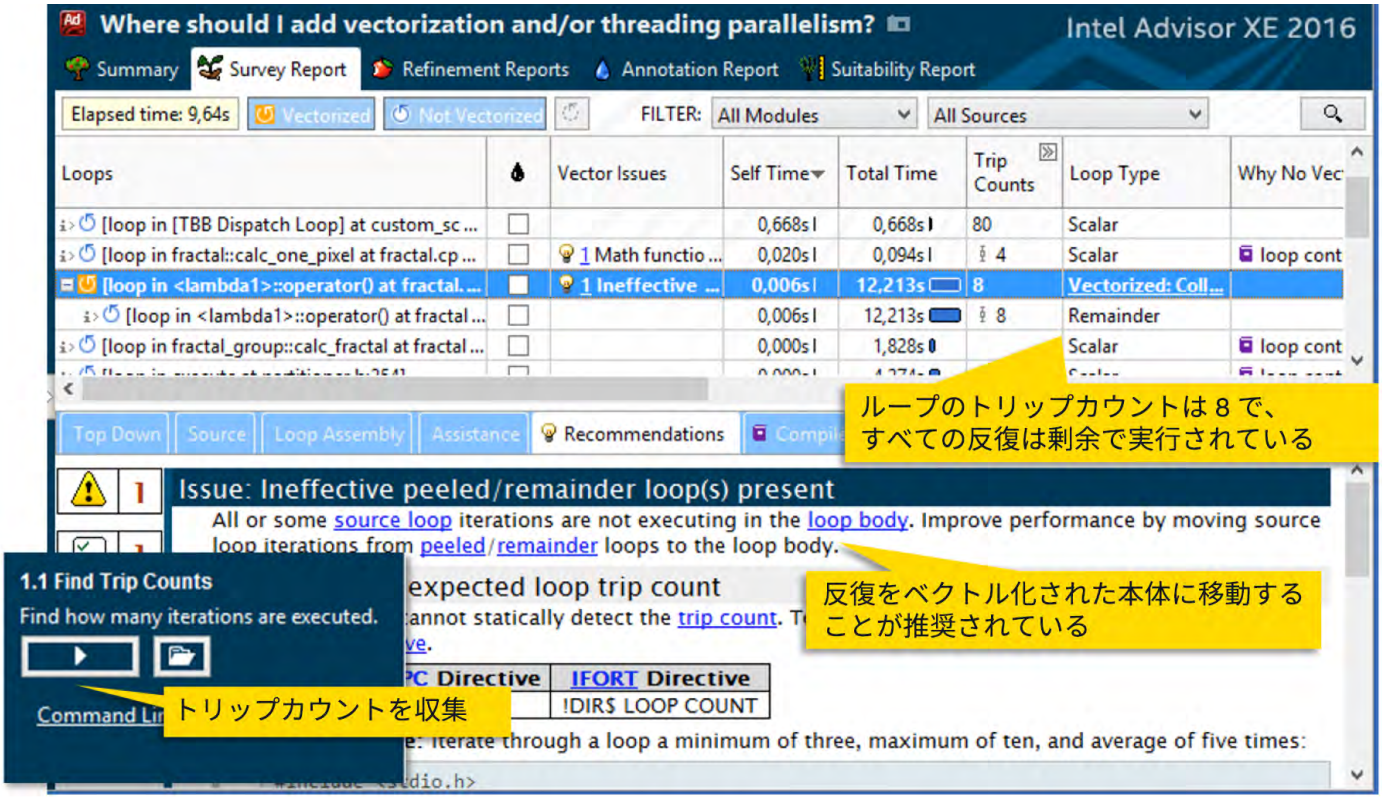
18 OpenMP* 4 を使用して関数をベクトル対応にする

次の調査プロファイルは、ループ内の剰余のみを表示しています。[Filter] バーの小さなボタンをクリックして、ループのフィルター「非ゼロ回のみ」を解除します (**図 19**)。ループのピール、本体、剰余部分が表示されます。これは、コンパイラーは 3 つの部分をすべて生成しているにもかかわらず、実際には剰余部分のみ実行されていることを意味します。



19 ループは剰余のみ実行

[Recommendations] タブでは、ループの既存の構造が問題であり、より多くの反復を本体に移動することが推奨されています。最初に、トリップカウント解析を実行して、ループの実際のトリップカウントを確認します (**図 20**)。



20 トリップカウント解析

ループには 8 つの反復しかなく、すべて剰余部分になることが分かりました。現在のベクトル長は、関数をベクトル対応にした後に可能になったコンパイラーの最適化ダブルポンプにより 16 です。ベクトル本体は少なくとも 16 の反復が必要であるため、このループはベクトル化された部分を実行するには小さすぎます。

該当ループはインテル® TBB の **parallel_for** ループから呼び出され、反復回数は粒度パラメーターにより定義されています。インテル® TBB の粒度を増やして、スレッドレベルの並列処理をより粗粒度にします。この変更により、SIMD ループで行う作業を増やせます (**図 21**)。

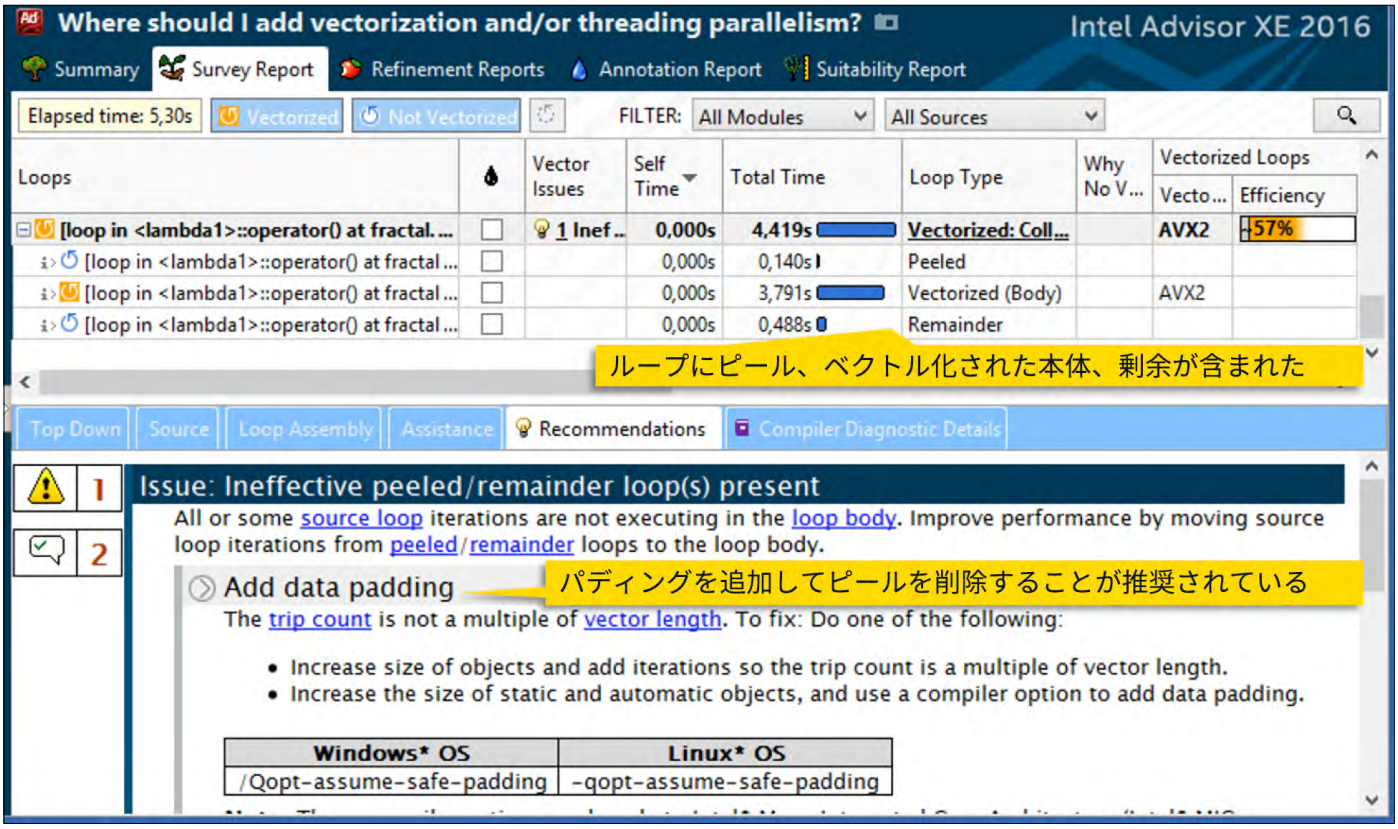
```
tbb::parallel_for(tbb::blocked_range2d<int>(y0, y1, inner_grain_size, x0, x1, inner_grain_size),
    [&] (tbb::blocked_range2d<int> &r){
    int x0 = r.cols().begin(),
        y0 = r.rows().begin(),
        x1 = r.cols().end(),
        y1 = r.rows().end();
    drawing_area area(off_x + x0, off_y + y0, x1 - x0, y1 - y0, dm);

    const int delta_x = x1 - x0;
    const int delta_y = y1 - y0;
    int delta_y2;
    int idx = 0;
    int status;

    color_t fractal_data_array[delta_x][delta_y];

    for (int x = x0; x < x1; ++x) {
        for (int y = y0; y < y1; ++y) {
            fractal_data_array[x - x0][y - y0] = calc_one_pixel(x, y, \
                tmp_max_iterations, tmp_size_x, tmp_size_y, tmp_magn, tmp_cx, tmp_cy, 255);
        }
    }
```

inner_grain_size は 8、128 に増やす

反復回数はインテル® TBB の parallel_for ループの粒度で制御されている

21 反復回数を増やす

反復回数を増やした後、ループは 3 つの部分 (ピール、本体、剰余) をすべて実行するようになりました。最終的に、CPU 時間は 12.5 秒から 4.4 秒に短縮され、大幅なパフォーマンス・ゲインを達成できました。しかし、インテル® Advisor XE は、さらにデータレイアウトを変更してピールを削除することを推奨しています (**図 22**)。



22 ピール / 本体 / 剰余ループ

ピールを削除するには、コードがインテル® Xeon® プロセッサーとインテル® Xeon Phi™ コプロセッサー (512 ビット・ベクトル・レジスター) の両方で適切に動作するように、データを 64 バイトでアライメントします。この変更により、キャッシュ使用率も向上し、キャッシュラインのフォルス・シェアリングの可能性が排除されました。

```
__declspec(align(64)) color_t fractal_data_array[delta_x][((delta_y - 1) / 16 + 1) * 16]; // aligned data

for (int x = x0; x < x1; ++x) {
    #pragma vector always aligned // force alignment
    for (int y = y0; y < y1; ++y) {
        fractal_data_array[x - x0][y - y0] = calc_one_pixel(x, y, \
            tmp_max_iterations, tmp_size_x, tmp_size_y, tmp_magn, tmp_cx, tmp_cy, 255);
    }
}
```

Summary | Survey Report | Refinement Reports | Annotation Report | Suitability Report

Elapsed time: 5,27s | Vectorized | Not Vectorized | FILTER: All Modules | All Sources

Loops		Vector Issues	Self Time	Total Time	Trip Counts	Loop Type	Why No Vectori...	Vectorized Loops	
								Vecto...	Efficiency
[loop in tbb::tbb:: at arena.cpp:88]			0,000s	4,379s		Scalar			
[loop in <lambda1>::operator() at fractal...		1 Ine...	0,000s	4,439s	7; 15	Vectorized: Coll...		AVX2	86%
[loop in <lambda1>::operator() at fracta...			0,000s	3,875s	7	Vectorized (Body)		AVX2	
[loop in <lambda1>::operator() at fracta...			0,000s	0,564s	15	Remainder			

ピールは削除されたが 15 回の反復が
スカラー剰余で実行されている

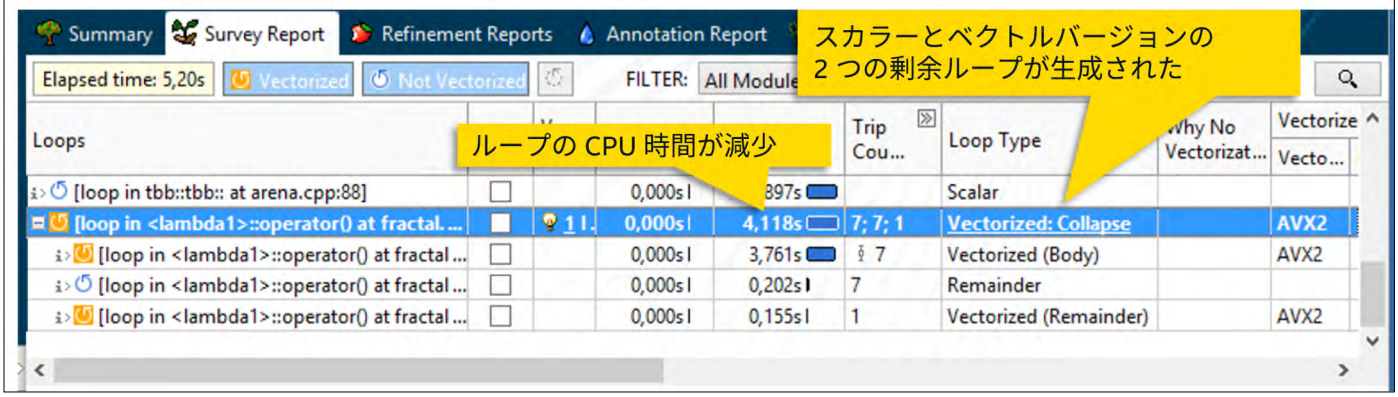
ループ SIMD の効率

23 データをアライメントしてピールを削除

変更後、ピール部分は表示されないようになりました (**図 23**)。ループの効率は 86 パーセントで、トリップカウント解析の実行後に更新されました。最初の効率はコンパイラーの推定に基づいていましたが、トリップカウントが利用可能になると、効率は実際の動的プロファイルに基づいて計算されます。

しかし、15 の反復がまだスカラー剰余で実行されています。**`#pragma vector`** 宣言子に **`vecremainder`** オプションを追加して、剰余ループの部分的なベクトル化を行います。ループのもう 1 つの部分 (ベクトル化された剰余) が生成され、スカラー部分の反復は 7 になります。この変更後、CPU 時間は大幅に短縮されました (**図 24**)。

```
__declspec(align(64)) color_t fractal_data_array[delta_x][((delta_y - 1) / 16 + 1) * 16]; // aligned data

for (int x = x0; x < x1; ++x) {
    #pragma vector always aligned vecremainder // force remainder vectorization
    for (int y = y0; y < y1; ++y) {
        fractal_data_array[x - x0][y - y0] = calc_one_pixel(x, y, \
            tmp_max_iterations, tmp_size_x, tmp_size_y, tmp_magn, tmp_cx, tmp_cy, 255);
    }
}
```



24 剰余をベクトル化

最適化の結果

サンプルは、次の構成のラップトップ上でテストされました。

- CPU: インテル® Core™ i5-4300U プロセッサー
- OS: Windows* 8 64 ビット
- コンパイラー : インテル® Parallel Studio XE 2016 Beta Update 1 Composer Edition
- インテル® Advisor XE 2016 Beta Update 3

SIMD 最適化の後、サンプルの実行時間はオリジナルの 4.45 秒から 1.35 秒に短縮され、3.2 倍のスピードアップが達成されました。

まとめ

コードに**ベクトル化**を追加するプロセスは気が遠くなるような作業に思われるかもしれませんが、ハードウェアを最大限に活用するには、ベクトル化は避けては通れない道です。この新しいツールを利用することで、この作業は劇的に改善されます。この記事で説明した、順序に沿ったアプローチを採用することにより、プログラムをベクトル化する作業を複数のステップに分割して、プロセスの各ステップで最適なツールを利用できます。

新しいインテル® Advisor XE は、2014 年の終わりにアルファプログラムに追加され、多くのユーザーによるテストで、非常に良好な結果が得られました。2015 年のベータプログラムでは、さらに多くのユーザーにこのツールを利用する機会が提供されました。インテル® Advisor XE は、多くの大規模 HPC アプリケーションでテストされ、優れた結果が得られています。

Software

THE PARALLEL UNIVERSE