インテル® ソフトウェア・ツールにおける浮動小数点結果の再現性

不確実性の払拭

Martyn Corden、Xiaoping Duan、および Barbara Perz
インテル コーポレーション ソフトウェア・テクニカル・コンサルティング・エンジニア

ほとんどの実数のバイナリー浮動小数点 (FP) 表現は不正確で、浮動小数点数を含むその演算結果には特有の不確実性があります。そのため、異なる条件下で計算を行うと、結果は予想される不確実さの範囲内で一貫していますが、異なる可能性もあります。通常は問題になりませんが、一部のコンテキストではより高い再現性が求められます (例: 品質保証、法的問題、機能安全要件など)。しかし、完全な再現性または高い再現性を達成するには、通常、パフォーマンスが犠牲になります。

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

再現性とは?

再現性の定義は人それぞれです。最も基本的なことは、同じプロセッサー上で、同じデータを使用して、同じ実行ファイルを繰り返し実行した場合、常に同一の結果が生成されることです。これは、繰り返し性または実行再現性と呼ばれます。結果が必ずしも決定的ではなく、再現性が自動的に提供されないことに驚いたり、ショックを受けるユーザーもいます。

再現性はまた、異なるプロセッサー・タイプをターゲットとし実行した場合、異なる最適化レベルでビルドした場合、あるいは異なる並列処理タイプやレベルで実行した場合に、同一の結果が生成されることを意味します。これは、条件付き数値再現性と呼ばれます。完全に再現性のある結果を得るために必要な条件はコンテキストに依存し、パフォーマンスが低下する可能性があります。

多くのソフトウェア・ツールは、デフォルトでは完全に再現性のある結果をもたらしません。

差異の原因

浮動小数点結果が異なる主な原因は、最適化です。最適化には、次のものが含まれます。

- ビルド時または実行時に特定のプロセッサーと命令セットを**ターゲット**にする
- **さまざまな**形式の並列処理

現代のプロセッサーでは、パフォーマンスの利点が非常に大きいため、大規模なアプリケーションでも最適化を行わないことはほとんどありません。精度が異なる原因として、次のようなものがあります。

- 数学関数や除算などの操作の**近似値が異なる**
- 中間結果の計算と格納に使用される**精度**
- **正規化されていない**非常に小さな値がゼロとして扱われる
- FMA (Fused Multiply Add) 命令などの**特殊命令の使用**

FMA のような特殊命令は、通常、乗算と加算を個別に行うよりも高い精度が得られますが、最終結果は変わる可能性があります。

FMA の生成は、インテル® アドバンスト・ベクトル・エクステンション 2 (インテル® AVX2) 以上の命令セットをターゲットとする場合に O1 以上で有効になる最適化です。言語標準ではカバーされていないため、コンパイラーは異なるコンテキストでは異なる最適化を適用する可能性があります (例: FMA をサポートする異なるプロセッサーに対して異なる最適化を適用することがあります)。

おそらく、差異の最も重要な原因は、特に並列アプリケーションの場合、操作の順序でしょう。異なる順序は数学的には等価かもしれませんが、有限精度演算では、丸め誤差により差異が生じたり、合計結果が異なることがあります。異なる結果が必ずしも精度が低いわけではありませんが、ユーザーは最適化されていない結果が正しいと見なすことがあります。

例えば、**図 1** に示すような、コンパイラーがパフォーマンス向上のために行う変換があります。

```
(x[i] + y) + z  →  x[i] + (y + z);

a*b + a*c →  a*(b+c)
```

1 **コンパイラーによる変換の例**

これまでに説明した最適化は、シーケンシャル・アプリケーションと並列アプリケーションに同様の影響を与えます。コンパイルされるコードでは、コンパイラー・オプションにより最適化を制御または抑止できます。

リダクション

リダクションは、結果が浮動小数点演算の順序に依存することを示す特に重要な例です。ここでは合計を例に説明しますが、ここで示す内容は、積、最大値、最小値などのほかのリダクション操作にも適用されます。合計操作の並列実装は、スレッドごと (OpenMP* など)、プロセスごと (MPI など)、SIMD レーンごと (ベクトル化) の部分和に分割します。これらの部分和はすべて安全に並列にインクリメントできます。**図 2** に例を示します。

```
//  original sequential
version

float Sum(const float A[],
int n)
{
    float sum=0;
    for (int i=0; i<n; i++)
        sum = sum + A[i];
    return sum;
}
```

```
float Sum( const float A[], int n, int nt ) {
  float sum=0.;                    // total sum
  float part_sum[nt]               // 1 partial sum per thread
  for (int it=0; it<nt; it++) part_sum[it]=0.;
  for (int it=0; it<nt; it++)  {  // loop over threads or ranks
      for (int j=it*n/nt; j<(it+1)*n/nt; j++)
          part_sum[it] += A[j];    //  partial sum within a thread
  }
  for (int it=0; it<nt; it++) sum += part_sum[it];
  return sum;
}                                  // parallel version
```

2 **リダクションを使用した並列合計処理**

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

2 つのケースでは、A の要素が加算される順序が大きく異なり、中間結果がマシン精度に丸められます。正の項と負の項の間に多くの相殺がある場合、最終結果への影響は驚くほど大きくなります。ユーザーは最初のシリアルバージョンが「正しい」結果と見なしがちですが、複数の部分和を使用する並列バージョンは、特に要素数が多い場合、丸め誤差の累積を減らし、有限精度の結果に近い結果を生成する傾向にあります。並列バージョンは演算も高速になります。

リダクションで再現性のある結果が得られるのか?

リダクションで再現性のある結果を得るためには、部分和の構成が常に同じでなければなりません。ベクトル化では、ベクトル長が常に同じでなければなりません。OpenMP* では、スレッド数が一定でなければなりません。**インテル® MPI ライブラリー**では、ランク数が常に同じでなければなりません。また、部分和を同じ一定の順序で合計する必要があります。ベクトル化ではこの処理が自動的に行われます。

OpenMP* によるスレッド化では、任意の順序で部分和を結合できます。インテルの実装では、スレッド数が小さい場合 (**インテル® Xeon® プロセッサー**では 4 未満、**インテル® Xeon Phi™ プロセッサー**では 8 未満)、デフォルトは先着順です。部分和が一定の順序で合計されることを確実にするには、環境変数 **`KMP_DETERMINISTIC_REDUCTION=true`** を設定して、静的スケジュール (デフォルトのスケジュール・プロトコル) を使用します。

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) は、動的スケジュールを使用するため、**`parallel_reduce()`** メソッドは実行再現性のある結果を生成しません。代わりに、**`parallel_deterministic_reduce()`** メソッドがサポートされています。このメソッドは、部分和を計算する固定タスクと、それを結合するための固定の順序付けされたツリーを作成します。動的スケジューラーは、依存関係を維持したまま、動的にタスクをスケジュールできます。これにより、同じ環境で実行再現性のある結果を生成できるだけでなく、ワーカースレッドの数が変わっても確実に再現性のある結果が得られます。(OpenMP*標準では、固定ツリーに基づく類似のリダクション操作を提供していませんが、OpenMP* タスクと依存性を利用して記述できます。)

インテル® MPI ライブラリーでは、プロセッサー・ノード間の MPI ランクの分散状況に応じて、部分和の結合順序を最適化できます。再現性のある結果を得る唯一の方法は、トポロジーを意識しないリダクション・アルゴリズムの制限されたセットから選択することです (以下を参照)。

我々が検証したすべての例では、並列またはベクトル化されたリダクションの結果は、通常、シーケンシャル・リダクションの結果とは異なりました。これが許容できない場合は、単一のスレッド、プロセス、または SIMD レーンでリダクションを実行する必要があります。単一のSIMD レーンで実行するには、リダクション・ループが自動でベクトル化されないように **`/fp:precise`** (Windows*) または **`-fp-model precise`** (Linux* および macOS*) を指定してコンパイルします。

インテル® コンパイラー

異なる実行、異なる最適化レベル、および同じアーキテクチャーの異なるプロセッサー・タイプで最高の再現性を実現するには、ハイレベルのオプション **/fp:consistent** (Windows*) または **-fp-model-consistent** (Linux* および macOS*) を推奨します。これは、FMA 生成を無効にする **/Qfma-** (**-no-fma**)、すべてのプロセッサー・タイプで同じ結果を生成する実装に数学関数を制限する **/Qimf-arch-consistency:true** (**-fimf-arch-consistency=true**)、および結果を変える可能性のあるその他のコンパイラーによる最適化を無効にする **/fp:precise** (**-fp-model-precise**) を指定した場合と同じです。

この再現性は、パフォーマンスを低下させます。どの程度低下するかはアプリケーションに依存しますが、一般に約 10% 低下します。通常、浮動小数点リダクションや超越数学関数呼び出しを含む、多くのベクトル化可能なループを持つ計算集約型アプリケーションが最も大きな影響を受けます。ベクトル呼び出しだけでなく、数学関数のスカラー呼び出しにも SVML (Short Vector Mathematical Library) ライブラリー関数を使用し、数学関数を含むループの自動ベクトル化の一貫性と再有効化を保証する **/Qimf-use-svml** (**-fimf-use-svml**) オプションを追加することでこの影響を軽減できることがあります。

デフォルトの **/fp:fast** (**-fp-model fast**) は、再現性に関係なくコンパイラーに最適化を許可します。同じプロセッサー上で、同じデータを使用して、同じ実行ファイルを繰り返し実行すると同じ結果が生成される、繰り返し性のみが必要な場合は、**/Qopt-dynamic-align-** (**-qno-opt-dynamic-align**) を指定して再コンパイルするだけで済む可能性があります。このオプションは、実行時にデータ・アライメントをテストするピールループの生成のみを無効にし、前述の **/fp** (**-fp-model**) オプションと比較するとパフォーマンスへの影響がわずかです。

異なるコンパイラーとオペレーティング・システムでの再現性

ほとんどの数学関数の結果には一般的に許容されている要件がないため、異なるコンパイラーとオペレーティング・システムによって再現性が制限されます。数学関数の最終的な基準 (例えば、正確な丸めを必要とするなど) を遵守すれは一貫性は向上しますが、パフォーマンスが大幅に低下します。

現在、Windows* や Linux* などの異なるオペレーティング・システムをターゲットとするコードの結果の再現性を体系的にテストする手段はありません。**/Qimf-use-svml** と **-fimf-use-svml** オプションは、数学関数を含むループのベクトル化に関連する既知の差異の原因に対応し、Windows* と Linux* の両方で浮動小数点結果の一貫性を向上するために推奨されます。

インテル® コンパイラーの異なるメジャーバージョンでビルドされたアプリケーションの間で一貫性を保証する手段はありません。改善された数学ライブラリー関数の実装では、結果の精度は向上するかもしれませんが、以前の実装とは異なる結果になる可能性があります。**/Qimf-precision:high** (**-fimf-precision=high**) オプションは、このような差分を軽減します。同様に、インテル® コンパイラーでビルドされたアプリケーションと他社製コンパイラーでビルドされたアプリケーションの間で再現性を保証することはできません。**/fp:consistent** (**-fp-model consistent**) などのオプションや他社製コンパイラーの同等のオプションは、コンパイル済みコードの結果の差異を軽減するのに役立ちます。可能な場合は、両方のコンパイラーで同じ数学ランタイム・ライブラリーを使用すると良いでしょう。

インテル® マス・カーネル・ライブラリー (インテル® MKL)

インテル® マス・カーネル・ライブラリー (インテル® MKL) は、OpenMP* またはインテル® スレッディング・ビルディング・ブロック (インテル® TBB) により内部でベクトル化またはスレッド化されている、線形代数、高速フーリエ変換、スパースソルバー、統計解析、およびその他のドメイン向けの高度に最適化された関数を提供します。デフォルトでは、同じプロセッサーで繰り返し実行した場合、最適化された関数内の操作の順序が変わるため同一の結果を得ることができません。インテル® MKL 関数は、実行時にプロセッサーを検出して、そのプロセッサー向けに最適化されたコードパスを実行するため、異なるプロセッサーで実行すると結果が異なる可能性があります。この問題を回避するため、インテル® MKL は条件付き数値再現性を実装しています。次の条件を満たす必要があります。

- インテル® TBB ではなく OpenMP* ベースのインテル® MKL バージョンを使用します。
- スレッド数を一定にします。
- 静的スケジューリングを使用します (デフォルトの **OMP_SCHEDULE=static**)。
- アクティブなスレッド数の動的調整を無効にします (デフォルトの **OMP_DYNAMIC=false** と **MKL_DYNAMIC=false**)。
- 同じオペレーティング・システムとアーキテクチャーを使用します (例: インテル® 64 アーキテクチャー・ベースの Linux*)。
- 同じマイクロアーキテクチャーを使用するか、最小マイクロアーキテクチャーを指定します。

最小マイクロアーキテクチャーは、関数またはサブルーチンを呼び出すか (例: **mkl_cbwr_set** (**MKL_CBWR_AVX**))、ランタイム環境変数を設定して (例: **MKL_CBWR_BRANCH=MKL_CBWR_AVX**) 指定します。これにより、インテル® AVX2 やインテル® AVX-512 などのインテル® AVX 以降の命令セットをサポートするすべてのインテル® プロセッサーで一貫した結果が得られます。ただし、より高度な命令セットをサポートするプロセッサーではパフォーマンスが低下します。引数 **MKL_CBWR_COMPATIBLE** は、同じアーキテクチャーのインテル® プロセッサーおよび互換プロセッサーで一貫した結果をもたらします。引数 **MKL_CBWR_AUTO** は、実行時に検出されたプロセッサーに対応するコードパスが実行されるようにします。そして、そのプロセッサーで繰り返し実行した場合に同じ結果が生成されることを保証します。ただし、ほかのプロセッサー・タイプでは結果が異なります。ランタイム・プロセッサーが指定された最小マイクロアーキテクチャーをサポートしていない場合、実行ファイルは実行できますが、**MKL_CBWR_AUTO** を指定した場合と同様に、実際のランタイム・マイクロアーキテクチャーに対応するコードパスが実行されます。結果は、警告なしに、ほかのプロセッサーの結果と異なる可能性があります。

命令セットを制限することで、計算集約型のインテル® MKL 関数では、パフォーマンスへの影響を大幅に軽減できることがあります。**表 1** は、**インテル® Xeon® スケーラブル・プロセッサー**上で、異なる最小マイクロアーキテクチャーを選択した場合の DGEMM 行列-行列乗算の相対パフォーマンスを示しています。

表 1. インテル® Xeon® スケーラブル・プロセッサーで DGEMM を実行した場合の命令セット・アーキテクチャー (ISA) への影響

ターゲット ISA	予測される相対パフォーマンス
`MKL_CBWR_AUTO`	1.0
`MKL_CBWR_AVX512`	1.0
`MKL_CBWR_AVX2`	0.50
`MKL_CBWR_AVX`	0.27
`MKL_CBWR_COMPATIBLE`	0.12

パフォーマンス結果は 2018 年 9 月 6 日現在のインテル社内での測定値であり、すべての公開済みセキュリティー・アップデートが適用されていない可能性があります。詳細は、システム構成を参照してください。絶対的なセキュリティーを提供できる製品はありません。性能に関するテストに使用されるソフトウェアとワークロードは、性能がインテル® マイクロプロセッサー用に最適化されていることがあります。SYSmark* や MobileMark* などの性能テストは、特定のコンピューター・システム、コンポーネント、ソフトウェア、操作、機能に基づいて行ったものです。結果はこれらの要因によって異なります。製品の購入を検討される場合は、他の製品と組み合わせた場合の本製品の性能など、ほかの情報や性能テストも参考にして、パフォーマンスを総合的に評価することをお勧めします。システム構成: インテル® Xeon® スケーラブル・プロセッサーで DGEMM 実行を実行した場合の命令セット・アーキテクチャー (ISA)。詳細については、**www.intel.com/benchmarks** (英語) を参照してください。

インテル® MPI ライブラリー

次の条件を満たす場合、インテル® MPI ライブラリーを使用する結果は再現性があります。

- **コンパイル済みコードとライブラリー呼び出し**がコンパイラーとライブラリーの再現性条件を満たす場合
- **MPI とクラスター環境に変更がない場合** (ランク数とプロセッサー・タイプを含む)

一般に、総和やリダクションなどの集合操作は、環境の小さな変化に対して最も敏感です。集合操作の多くの実装は、クラスターノード間の MPI ランクの分散状況に応じて最適化されており、操作の順序が変更されたり、結果に差異が生じることがあります。インテル® MPI ライブラリーは、条件付き数値再現性をサポートしており、ノード間のランクの分散状況が変更されても、アプリケーションは同じバイナリーで再現性のある結果を得られます。それには、`I_MPI_ADJUST_` 環境変数を使用して、トポロジーを意識しない (つまり、ノード間のランクの分散状況に応じて最適化しない) アルゴリズムを選択する必要があります。

- `I_MPI_ADJUST_ALLREDUCE`
- `I_MPI_ADJUST_REDUCE`
- `I_MPI_ADJUST_REDUCE_SCATTER`
- `I_MPI_ADJUST_SCAN`
- `I_MPI_ADJUST_EXSCAN`
- ほか

例えば、インテル® MPI ライブラリー・デベロッパー・リファレンスには、11 種類の `MPI_REDUCE()` の実装があります。そのうちの最初の 7 つを**表 2** に示します。

表 2. 異なるランクの分散状況での MPI_REDUCE() の結果の比較

	ランク分散 アルゴリズム	0246 node1 1357 node2	すべての node1	0145 node1 2367 node2	0123 node1 4567 node2
1	Shumilin				
2	二項				
3	トポロジーを意識した Shumilin				
4	トポロジーを意識した二項				
5	Rabenseifner				
6	トポロジーを意識した Rabenseifner				
7	Knomial				

インテル® Core™ i5-4670T プロセッサー @ 2.30GHz、4 コア、8GB メモリーベースの 2 つのノード。1 つは Red Hat* EL 6.5、もう一方は Ubuntu* 16.04 を実行。The Parallel Universe 21 号の「インテル® MPI ライブラリーの条件付き再現性」のサンプルコードを使用 (記事の最後にある「参考文献」を参照)。

表 2 は、選択した **MPI_REDUCE()** 実装のサンプルプログラムを、8 つの MPI ランクを 2 つのクラスターノードに 4 つの異なる方法で分散して実行した結果の比較です。測定された 5 つの異なる結果に応じて色分けしています。結果の差は非常に小さく、精度の限界に近いものですが、大規模な計算では小さな差分が取り消しによって増幅されることがあります。トポロジーに依存しない実装は、ノード間のランクの分散状況に関係なく同じ結果を生成しますが、トポロジーを意識した実装では結果が異なります。**MPI_REDUCE** のデフォルトの実装 (ここには記載されていません) は、ワークロードとトポロジーに依存するアルゴリズムで構成されており、ノード間のランクの分散状況に応じて結果が異なります。

結論

インテル® ソフトウェア開発ツールは、明確に定義された条件下で再現性のある浮動小数点結果を取得する方法を提供します。

参考文献

1. 「**インテル® コンパイラーの浮動小数点演算における結果の一貫性**」
2. 『**インテル® MKL 2019 for Linux* デベロッパー・ガイド**』(英語) の「Obtaining Numerically Reproducible Results (数値再現性のある結果を得る)」セクション
3. The Parallel Universe 21 号の「**インテル® MPI ライブラリーの条件付き再現性**」
4. 『**インテル® MPI ライブラリーのチューニング: 基本的な手法**』(英語) の「Tuning for Numerical Stability (数値安定性のためのチューニング)」セクション
5. 『**インテル® C++ コンパイラー・デベロッパー・ガイドおよびリファレンス**』(英語) と『**インテル® Fortran コンパイラー・デベロッパー・ガイドおよびリファレンス**』(英語) の「Floating-Point Operations (浮動小数点演算)」セクション

BLOG HIGHLIGHTS

ドメイン・エキスパートとチューニング・エキスパートの間のギャップを解消

Henry A. Gabb インテル コーポレーション シニア主席エンジニア

2006 年にカリフォルニア大学バークレー校の並列コンピューティング研究所は、並列処理が広く普及することでドメイン・エキスパートとチューニング・エキスパートの間の課題の分業はさらに大きくなると予測しました (図 1)。一方には問題を解こうとするあらゆる分野のユーザーがおり、彼らのコンピューター・バックグラウンドはさまざまです。彼らは問題を解くために必要なコーディングを最小限に抑え、ビジネス上の意思決定、研究論文、工学的設計などのより大きなタスクに集中したいと考えています。コードのチューニングは、パフォーマンスのボトルネックが目標達成を妨げる場合にのみ考慮されます。もう一方には、その対極と言えるチューニング・エキスパート (内部では通称忍者プログラマーと呼ばれている) がおり、彼らは大規模なアプリケーションにおいて重要ではないコード領域から最大限のパフォーマンスを引き出そうとします。

この記事の続きはこちら (英語) でご覧になれます。>

Software

システムと IoT アプリケーションを高速化

インテル® System Studio 2019 の新機能

新しい電力およびパフォーマンス・ツールにより、開発サイクルを短縮して迅速に市場へ投入

高度なクラウドコネクターと 400 を超えるセンサーを利用可能。

パフォーマンス・ボトルネックを素早く特定、電力使用を軽減、ほか

無料のダウンロード >

https://www.isus.jp/intel-system-studio/

[1] 性能に関するテストに使用されるソフトウェアとワークロードは、性能がインテル® マイクロプロセッサー用に最適化されていることがあります。SYSmark* や MobileMark* などの性能テストは、特定のコンピューター・システム、コンポーネント、ソフトウェア、操作、機能に基づいて行ったものです。結果はこれらの要因によって異なります。製品の購入を検討される場合は、他の製品と組み合わせた場合の本製品の性能など、ほかの情報や性能テストも参考にして、パフォーマンスを総合的に評価することをお勧めします。詳細については、www.intel.com/benchmarks (英語) を参照してください。

詳細は、software.intel.com/en-us/intel-parallel-studio-xe/details#configurations (英語) を参照してください。

コンパイラーの最適化に関する詳細は、最適化に関する注意事項 (software.intel.com/en-us/articles/optimization-notice#opt-jp) を参照してください。